realtimepublishers.com™

The Administrator Shortcut Guide™ To

VBScripting for Windows

Don Jones

Introduction

By Sean Daily, Series Editor

Welcome to *The Administrator Shortcut Guide to VBScripting for Windows*!

The book you are about to read represents an entirely new modality of book publishing and a major first in the publishing industry. The founding concept behind Realtimepublishers.com is the idea of providing readers with high-quality books about today's most critical IT topics—at no cost to the reader. Although this may sound like a somewhat impossible feat to achieve, it is made possible through the vision and generosity of corporate sponsors who agree to bear the book's production expenses and host the book on its Web site for the benefit of its Web site visitors.

It should be pointed out that the free nature of these books does not in any way diminish their quality. Without reservation, I can tell you that this book is the equivalent of any similar printed book you might find at your local bookstore (with the notable exception that it won't cost you $30 to $80). In addition to the free nature of the books, this publishing model provides other significant benefits. For example, the electronic nature of this eBook makes events such as chapter updates and additions, or the release of a new edition of the book possible to achieve in a far shorter timeframe than is possible with printed books. Because we publish our titles in "real-time"—that is, as chapters are written or revised by the author—you benefit from receiving the information immediately rather than having to wait months or years to receive a complete product.

Finally, I'd like to note that although it is true that the sponsor's Web site is the exclusive online location of the book, this book is by no means a paid advertisement. Realtimepublishers is an independent publishing company and maintains, by written agreement with the sponsor, 100% editorial control over the content of our titles. However, by hosting this information, the sponsor has set itself apart from its competitors by providing real value to its customers and transforming its site into a true technical resource library—not just a place to learn about its company and products. It is my opinion that this system of content delivery is not only of immeasurable value to readers, but represents the future of book publishing.

As series editor, it is my raison d'être to locate and work only with the industry's leading authors and editors, and publish books that help IT personnel, IT managers, and users to do their everyday jobs. To that end, I encourage and welcome your feedback on this or any other book in the Realtimepublishers.com series. If you would like to submit a comment, question, or suggestion, please do so by sending an email to feedback@realtimepublishers.com, leaving feedback on our Web site at www.realtimepublishers.com, or calling us at (707) 539-5280.

Thanks for reading, and enjoy!

Sean Daily

Series Editor

Copyright Statement

Chapter 1: Introduction to VBScript

Have you ever run around to each computer on your network checking to see whether a particular patch was installed? Have you spent hours creating new user accounts and Exchange mailboxes for a batch of new users? Have you and a coworker walked around powering off client computers at the end of the day? In addition to these mundane duties, you've probably performed any number of tedious, time-consuming, repetitive tasks. Have you ever wondered if there was an easier way?

Practically any Windows administration task can be automated. There are software developers that spend their careers creating tools to automate Windows administration. Even Microsoft provides Windows administration automation tools with command-line utilities such as Cacls.exe, which helps automate the process of changing NTFS file permissions. But as a Windows administrator, you probably think you don't have time to sit around programming your own automation tools. That's where you're wrong: VBScript offers a powerful, easy to understand scripting language that is practically tailor-made for Windows administration.

This guide will provide the basics of VBScript. Rather than attempt to show you every nook and cranny of VBScript—because you're probably more interested in just getting the job done, this guide will show you everything you need to know to get started with VBScript. In addition, there will be plenty of sample scripts to give you a little jump start.

> 🕮 When you're ready for more, check out http://www.ScriptingAnswers.com. You'll find dozens more sample scripts, tools and utilities, reviews of scripting-related products, tutorials, and tons more, all free and all firmly focused on the life of a Windows administrator.

What is VBScript?

Simply put, VBScript is a programming language. It's fairly easy to learn because its commands are all common English words (well, most of them are common). Unlike languages such as C++ or JavaScript, VBScript isn't case-sensitive, and you don't have to use much in the way of special formatting in your scripts. These qualities make VBScript pretty forgiving of new (or just lazy) programmers, and it makes the language a bit easier and more enjoyable to learn. The following example shows a simple VBScript program:

```
Dim sMyName
sMyName = InputBox("Type your name")
MsgBox "Hello, " & sMyName
```

Let's take a moment to examine this simple script. First, the script tells VBScript that you plan to use a *variable* named sMyName. A variable is exactly what you remember from high school algebra: it's a name that holds a changing value. At the beginning of the script, sMyName doesn't contain any value; you're simply announcing to VBScript that you plan to use the variable.

Next, the script assigns a value to sMyName. You can see that a value is being assigned because sMyName is on the left side of an equal sign (=). In algebra, writing something like x = 5 assigned the value 5 to the variable x; VBScript works the same way. In this case, however, the script is not assigning a literal value like 5; instead, the script is assigning the result of a *function*. The function in this example is InputBox(). This function displays a dialog box in which the user can type something. The message in the dialog box is "Type your name." What the user inputs will be placed into the variable sMyName.

Next, the script uses a statement called MsgBox. This function displays a message box, or dialog box, containing "Hello" and whatever value is stored in sMyName. Thus, if you had typed "Joe" into the InputBox, the message box would display "Hello, Joe."

To try scripting this simple example on your own, right-click on your desktop, and select New, Text document. Rename the new document to Test.vbs; when Windows warns you that you're changing the filename extension, click Yes to tell Windows that you know what you're doing and to go ahead and change the extension. Open Notepad.exe, drag Test.vbs into Notepad, type the three lines of code that the example shows, and save the file. Finally, double-click Test.vbs to execute the program.

You've just written your first script! If you can't get this script to run on your system, scripting might be disabled on your system or Windows might be configured not to show filename extensions. To solve these problems, enable scripting and try downloading and installing the Windows Script Host (WSH), version 5.6, from http://www.microsoft.com/scripting.

WSH is the software that actually executes your scripts.

Once you've have the script working, you're an official scripter. Believe it or not, you've already learned about several of VBScript's most important concepts:

- **Variables—**Variable are names that hold a changing value. That value can be a string of characters (such as a name), a date or time, a number, a true/false value (called a *Boolean*), and so forth. VBScript is flexible, you can use string, numeric, or whatever-type value in a variable.
- **Operators—**The equal sign in the example script is an *assignment operator*. As I already explained, it's responsible for placing the output of InputBox() into sMyName. Other operators perform mathematical operations:
  - + handles addition
  - - for subtraction
  - * handles multiplication
  - / denotes division

  VBScript provides many additional math functions, but you're not likely to need the cosine or tangent of some number in your Windows administrative tasks; thus, you can probably ignore all but the basic four math operators.

- **Functions—**I'll spend more time on functions in the next section, but you've already seen how functions work. They can (but don't always) accept an input value (in this case, it was the prompt to "Type your name"). They always return some value, which in this case, is whatever the user inputs. The function's input parameters are enclosed in parentheses, and the function's output is often assigned to a variable or used as the input parameter to another function.
- **Statements—**The last line of the example script is a statement. Notice that, like a function, it accepts a parameter. However, a statement doesn't enclose that parameter in parentheses. The reason is that the statement, unlike the function, doesn't return any value; it simply does something, which, in this case, is to display a message box.

Most scripts are this simple. Sure, they might be *longer*, but they're not logically more complex than this simple example.

Functions and Statements

You've already been introduced to one function, InputBox(), so you're ready to start exploring the VBScript documentation. To do so, point Microsoft Internet Explorer (IE) to http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vtoriVBScript.asp.

> I specified Internet Explorer specifically rather than generically referred to any Web browser. The reason is that the documentation site works better with IE.

Click VBScript Language Reference, click Functions, and click InputBox. The official VBScript documentation says that the syntax of the InputBox function is:

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

What this information means is that the InputBox() function can accept as many as seven parameters. All but the first is enclosed in square brackets, which means they're optional. The first parameter, *prompt*, is a string that is displayed in the InputBox. You can specify additional parameters to give the InputBox a title or to provide a default input value for users who are too lazy to type. If you don't want the InputBox centered, you can specify a starting position using X and Y coordinates. If you've written a help file for your script, you can specify the help file and the InputBox's *context link ID number* (that is the ID number that tells Windows which help file topic to load if someone presses F1 while the InputBox is displayed).

These parameters must *always* fall in this order. You can't skip any. For example, if you want to use the *prompt* and *default* parameters, but don't care about *title*, you would use a script similar to the following example:

```
sMyName = InputBox("Type your name", , "Joe")
```

Between the two commas is the *title* parameter. Although the title has been left out, you must include the comma after the *title* parameter so that VBScript realizes "Joe" is the third parameter.

Exploring Functions

If you're on the InputBox function page and you click the Back button in your browser, you'll be confronted with an alphabetical list of every function VBScript knows about. As an administrator, you will probably use only ten percent of these functions. The following list highlights functions that you will most likely use often:

- CDate()—Converts a value into a date
- CInt()—Converts a value into an integer
- CStr()—Converts a value into a string
- Date()—Returns the current system date
- DateAdd()—Adds and subtracts dates
- DateDiff()—Provides the difference (in days, weeks, or a specified metric) between two dates
- DatePart()—Returns a part of a date (for example, the year, day, month, and so on)
- GetObject()—Returns a reference to an automation object.

We'll explore GetObject in detail later in this chapter.

- InputBox()—As we've examined, displays a dialog box in which the user can type something
- InStr()—Tells you where one string (such as ar) can be found in another string (such as Mars; the answer is 2 because ar occurs at the second character of Mars)
- LCase()—Converts a string to lowercase
- Left()—Grabs the leftmost specified number of characters of a string
- Len()—Tells you the length of a strings
- MsgBox()—Is a function *and* a statement; as a function, it displays different buttons (such as Yes and No) and determines which button the user selected
- Now()—The current system date and time
- Replace()—Similar to InStr(), except Replace() locates any occurrence of one string within a second string and replaces those occurrences with a third string
- Right()—Returns the rightmost specified number of characters of a string
- Split()—Takes a delimited list (such as "one,two,three") and breaks it into an array of the list values
- Time()—Returns the current system time
- UCase()—Converts a string to uppercase

You'll find other functions to be useful later in your scripting career, but these twenty functions will be useful as you begin scripting.

> ☞ When you're trying to figure out how to do something in VBScript, browse the function list to see if anything looks likely. For example, if you need to write a script that converts user input to uppercase, you might browse the function list for something that starts with "upper" or "U." The first function under "U" is UBound(), which doesn't sound likely as a solution; the second function is UCase(), which sounds like it might be a winner. In addition, browsing can be a great way to find out more about the language.

Using Functions

So how do you use a function? Well, you've already seen InputBox() in action. Let's start with this function and create a script that also includes as many other common functions as possible (see Listing 1.1).

```
Dim sVar
sVar = InputBox("Go ahead, type something.","Test","Something")

MsgBox "The first letter is " & Left(sVar, 1)
MsgBox "and the last letter is " & Right(sVar, 1)
MsgBox "In uppercase it's " & UCase(sVar)
MsgBox "Today is " & Date()
MsgBox "Right now it is " & Now()
MsgBox "The second character is " & Right(Left(sVar,2),1)
```

Listing 1.1: An example script that uses many common functions.

Notice that the script that Listing 1.1 shows uses the ampersand (&) character. This character appends, or *concatenates*, two strings so that they appear to be one string.

Also look at the last line of code in which the script uses two functions together. The way to read these is to start with the inside-most function: Left(sVar,2), which will return the leftmost two characters of whatever is in sVar. Next is Right(*Left(sVar,2)*, 1); the outer function will return the rightmost one character of whatever the inner function output. Thus, if you start with the leftmost two characters, then take the rightmost one character, you end up with the second character in the string.

This example illustrates that functions can be nested within one another; however, it provides a difficult and roundabout way to reach the second character in the string. You could achieve the same result with the following code:

```
MsgBox "The second character is " & Mid(sVar, 2, 1)
```

This script will take sVar, start at the second character, and return a string of one character.

> 🕮 For more information about Mid(), look it up in the VBScript documentation.

Fancy Variables

The last section briefly mentioned *arrays* (there was a quick reference to the Split() function). An array is basically a list contained within a single variable; arrays are used in several administrative scripting situations. For example, consider the following script:

```
Dim sVar
sVar = "Hello"
```

sVar is not an array. It contains a single value, "Hello." Consider the next script:

```
Dim sVar
sVar = Split("One,Two,Three", ",")
```

sVar is now an array, containing three values: "One" is the first value, "Two" is the second, and "Three" is the third. The Split() function removed the commas when it split the list, using those commas as delimiters. The second parameter in Split() tells the script to use a comma—rather than some other character—as the delimiter for the list. To access the *elements* in an array, you use a script similar to the following:

```
MsgBox sVar(0)
MsgBox sVar(1)
MsgBox sVar(2)
```

Notice that the array starts numbering at zero, not one; thus, an array with three elements will have *indexes* ranging from zero to two. There are functions to tell you the index number of the array's last element—MsgBox UBound(sVar) would return "2" in this example.

You can make your own arrays, put data into individual elements, and so forth. You can also use *multi-dimensional arrays*, such as the following example:

```
Dim MyArray(5,1)
MyArray(0,0) = "Hello"
MyArray(0,1) = "There"
```

Think of a two-dimensional array like this example as a kind of ersatz Excel spreadsheet. The first dimension (with elements ranging from zero to five) is the spreadsheet's rows; the second (zero and one) are columns. You can have three-dimensional arrays, four-dimensional, and so forth.

Adding Logic

Once you've mastered variables and functions, you can begin enabling your scripts to think for themselves. For example, I mentioned earlier how you can use MsgBox as a function to ask yes/no questions. How would you program your script to handle a yes or a no individually? As the following script show, you use logic:

```
iResponse = MsgBox("Continue?", 4)
If iResponse = 6 Then
 MsgBox "Here we go!"
Else
 MsgBox "Aw, too bad."
End If
```

Notice the second parameter, which is the number 4. The VBScript documentation informs you that this is the correct number for a message box with Yes and No buttons.

> ☞ Did you catch how the second parameter of MsgBox() isn't included in quotation marks? VBScript uses double quotes to identify strings of characters; anything you want treated as a number doesn't need quotes.

Next comes an If…Then *construct* that has three parts:

- **If—**Some comparison is offered, in this case iResponse = 6. If iResponse does, in fact, equal six, the next line of code will be executed. Why six? The VBScript documentation for MsgBox() tells you that MsgBox() will return a 6 if the user clicks the Yes button. If the user clicks No, the result is a seven.
- **Else—**What if iResponse doesn't equal six? No problem, VBScript will start looking for other options, which is given in the Else portion of the construct. If the user clicks No, the lines of code following Else will execute.
- **End If—**When the user has made a selection and the script has executed the appropriate code, VBScript looks for the End If line.

Let's walk through the logic. If the user clicks Yes, you get a 6 back. VBScript will display "Here we go!" in a message box. The next line of code is Else; VBScript doesn't need an else because the original condition was true, so VBScript goes looking for End If.

If the user clicks No, you get back a 7. VBScript starts looking for other options. An ElseIf, which we haven't yet explored, an End If, or an Else. The first option that the script runs across is Else, so the script executes that code and displays the related message. What if you have more than a simple yes/no decision? Listing 1.2 provides an example for this type of scenario.

```
iResponse = MsgBox(“What do you want to do?”, 2)
If iResponse = 3 Then
 MsgBox “Abort!”
ElseIf iResponse = 4 Then
 MsgBox “Retry!”
ElseIf iResponse = 5 Then
 MsgBox “Ignore!”
Else
 MsgBox “What did you click?”
End If
```

Listing 1.2: Example script that illustrates more than a yes/no decision.

In Listing 1.2, the 2 in the MsgBox() function forces an Abort, Retry, and Ignore button to be displayed. According to the documentation, clicking those buttons will yield a value of 3, 4, or 5, respectively.

If the user clicks Abort, you get a 3, so the If evaluates to true and the first MsgBox is displayed. If not, VBScript looks for alternatives; it will first run across an ElseIf. If *that* evaluates to true, the second MsgBox is displayed. If not, the *next* ElseIf is examined; if it is not true either (I’m not sure how that could happen with only three buttons to choose from, but let’s imagine), VBScript goes with the final option: Else.

Choosing from a List of Possibilities

There is a slightly easier way to work with a large list of possible choices, called Select…Case. Listing 1.3 shows an alternative to the previous script.

```
iResponse = MsgBox(“What do you want to do?”, 2)
Select Case iResponse
 Case 3
  MsgBox “Abort!”
 Case 4
  MsgBox “Retry!”
 Case 5
  MsgBox “Ignore!”
 Case Else
  MsgBox “What did you click?”
End Select
```

Listing 1.3: Another example script that handles a list of possibilities.

In the script that Listing 1.3 shows, the Select statement tells VBScript which variable we will be examining, and each Case statement provides a different possible value for that variable. If the variable doesn’t contain any of the listed cases, then Case Else provides a final alternative.

> ☞ Notice how the lines of code are indented a bit within each construct? This technique makes it easier to keep track of which code is inside a construct, and to make sure you properly end each construct (with End Select in this example) at its completion. You don’t *need* to indent like this; VBScript doesn’t care. But your code will be easier to read and debug if you use this best practice.

Executing Code Again and Again and Again

If…Then and Select…Case are VBScript's two *logical constructs* (also called *conditional statements*). VBScript also has *looping constructs*, which are designed to execute a portion of code over and over. A practical application of a looping construct is working with Active Directory (AD), where you might have a script loop through each organizational unit (OU) in the domain and do something with it. A more fun example is to write a script that is a bit like a baby brother—cute, but not very useful, and slightly annoying:

```
Dim iVar
iVar = 1
Do Until iVar = 10
 MsgBox "Why?"
 iVar = iVar + 1
Loop
```

The script starts by assigning the value 1 to the variable iVar. Next, the script executes a Do Until…Loop construct. In this example, the code within the loop (the two indented lines) will execute until iVar contains a value of 10. Each time through the loop, a message box containing "Why?" will display. Then, iVar will be modified to contain a value equal to its current value, plus one. In other words, VBScript will add one to whatever iVar contains, then store the result right back in iVar. In this fashion, iVar will eventually reach ten, and the loop will stop.

> ☞ You might have noticed that this guide uses a *naming convention* for variables: If the variables are meant to contain strings, their names start with an "s;" if they're meant to contain integers, they start with an "i;" date variables start with a "d;" and so forth. Naming conventions don't affect how VBScript handles code—you could name variables "kkhlkj" without affecting the output of the script. The variable naming just makes it easier for *you* to remember the purpose of the variable.
>
> Microsoft has an official naming contention that you'll see in its scripts: "str" for strings, "int" for integers, and so on. You can make up your own system, or use someone else's system that makes sense to you.

The following script is a very minor variation from the previous script:

```
Dim iVar
iVar = 1
Do
 MsgBox "Why?"
 iVar = iVar + 1
Loop Until iVar = 10
```

Do you see the variation? iVar isn't being evaluated until the Loop statement. In practice, these two scripts will execute identically. However, consider the following variation:

```
Dim iVar
iVar = 10
Do Until iVar = 10
 MsgBox "Why?"
 iVar = iVar + 1
Loop
```

Notice that iVar is now being assigned a starting value of ten, which means no message boxes will ever be displayed, because the condition in the Do Until statement is already satisfied. Now, move the iVar evaluation back to the Loop statement:

```
Dim iVar
iVar = 10
Do
 MsgBox "Why?"
 iVar = iVar + 1
Loop Until iVar = 10
```

iVar isn't being evaluated until later—the message box will display *once*, because the loop will execute once before it gets around to checking the value of iVar. This difference is the only difference between the two: the location and timing of the variable evaluation. When Do appears by itself, the loop will always execute at least once; when Do has an Until clause, the loop will only execute if the Until clause's expression is false to begin with.

Let's flip the logic around a bit:

```
Dim iVar
iVar = 1
Do While iVar = 10
 MsgBox "Why?"
 iVar = iVar + 1
Loop
```

See the change? The script doesn't loop *until* iVar equals ten, it loops only *while* iVar equals ten. Because iVar is getting an initial value of one, the loop will never execute because the initial While clause is false. The clause can be moved to the Loop statement, as well:

```
Dim iVar
iVar = 1
Do
 MsgBox "Why?"
 iVar = iVar + 1
Loop While iVar = 10
```

Now the loop will execute once. When it reaches the Loop While statement, it will realize that iVar doesn't equal ten (it equals two, at that point), the loop will terminate, and VBScript will continue with whatever code follows.

Remember that these loops don't need to use a variable to control their execution; check out the following example:

```
Dim sResponse
Do While LCase(sResponse) <> "uncle"
 sResponse = InputBox("Say Uncle!")
Loop
```

After declaring the intention of using a variable named sResponse, the script enters a loop that will execute until sResponse contains the string "uncle". The script actually convert sResponse to lowercase using the LCase() function, and compares *that* to the string "uncle," allowing the user to type in mixed case and still get it right. Within the loop, an InputBox() function encourages the user to "Say Uncle!" and will pop up again and again until the user actually does so.

> ☞ Or, the user could press Ctrl+Break on the keyboard, breaking out of the script and terminating it. You can always terminate a VBScript by using this method.

Declaring Variables Carefully

Be aware that a single misplace keystroke can have repercussions. Carefully examine the example script and predict what will happen when it executes:

```
Dim sResponse
Do While LCase(sRespone) <> "uncle"
 sResponse = InputBox("Say Uncle!")
Loop
```

This script will loop *forever*, until someone presses Ctrl+Break, or the user reboots the machine. The reason is that the InputBox() result is being stored in sReponse, but the loop is examining the value of a different variable, *sRepone*. Oops. One reason this happens, other than lazy typing, is that VBScript doesn't *require* you to announce variables ahead of time by using the Dim statement. When VBScript runs across the sRespone variable, it creates the new variable and gives it the default value of nothing. Because that variable isn't being modified by the script, an *infinite loop* results.

You can guard against this sort of typo with a single statement at the start of each script:

```
Option Explicit
Dim sResponse
Do While LCase(sRespone) <> “uncle”
 sResponse = InputBox(“Say Uncle!”)
Loop
```

VBScript will see you declaring the variable sResponse, and think that all is in good order. When VBScript runs across the never-before-heard-of sRepone, it will throw an error (on line 3 of your script) indicating that you have an *undeclared variable*.

> ☞ Option Explicit is a highly recommended addition to any script that will help keep you out of scripting trouble.

Alternative Loops

VBScript contains a completely different type of loop called a For…Next loop. This loop type is mainly useful for repeating something a specific number of times rather than repeating something until a condition is true or false. The following script provides an example of a For…Next loop:

```
Dim iVar
For iVar = 1 To 10
 MsgBox iVar
Next
```

iVar is assigned an initial value of 1 by the For statement. Each time the script hits Next, the script increments iVar by the default increment of 1. When iVar reaches 11—out of the bounds specified by the For statement—the loop terminates.

You can modify the default increment value:

```
Dim iVar
For iVar = 1 To 10 Step 2
 MsgBox iVar
Next
```

Or, you can count backwards:

```
Dim iVar
For iVar = 10 To 1 Step -1
 MsgBox iVar
Next
```

Run these two short scripts to see what they do. Keep For…Next firmly in mind because you’re going to work with it more frequently as you begin working with objects.

Working with Objects

Believe it or not, you've actually learned everything an administrator needs to know about VBScript. The fact is that VBScript isn't a complicated language. However, what you've learned so far won't result in faster Windows administration. VBScript's real value as a scripting language isn't in its built-in capabilities. What makes VBScript powerful is its ability to use the Component Object Model (COM) objects that make up the Windows operating system (OS). VBScript sort of glues these objects together and makes them do interesting, useful things.

Objects generally represent specific OS functionality, such as Active Directory, the file system, the network, and so on. Objects have four primary characteristics that you need to be concerned with:

- **Properties—**Describe various attributes of an object
- **Methods—**Instruct an object to perform actions
- **Collections—**Some objects are comprised of multiple child objects, which are contained in collections.
- **Events—**Occur when an object does something, or encounters something, or when something happens to the object

Administrative scripts don't really need to use events all that often, so let's focus on properties, collections, and methods. To try and make this clearer, let's take a sample object: Car. The Car object has several properties, including ModelYear, EngineSize, Make, Model, and Color. This car is computerized, so you can *read* these properties and even *write* them. For example, you might set the ModelYear property to 2004, or set the Color property to Green. You could check the EngineSize property and have your script take a different action if the EngineSize was 4.0L instead of 2.5L.

The Car object also has a collection, named Tires. Each child of the Tires collection is, predictably enough, a Tire object. Each Tire object has its own properties, such as Position, Miles, Size, and so forth. If you wanted to see the number of miles on each tire, you might write a loop like this:

```
Dim oCar, oTire
Set oCar = CreateObject("Car")
MsgBox "Tire report for " & oCar.ModelYear & " " oCar.Model
For Each oTire In oCar.Tires
 MsgBox oTire.Position & " has " & oTire.Miles & " miles."
Next
```

> Don't bother typing in this script and trying to run it—there's no such object as Car. This script is just an example of how objects work.

First, VBScript is asked to create the Car object and assign it to the variable oCar. Notice that this assignment isn't like a normal value assignment. Although an equal sign is used, the Set statement is also required. CreateObject() is a built-in function, and Car is the registered class name for the Car object (for more information about creating objects, read the sidebar, "Creating Objects").

A message box displays a brief introduction along with the values of the car's ModelYear and Model properties ("Tire report for 2004 Wrangler"). Notice that the variable oCar is used to refer to the car, and that the properties are "attached" to that reference by a period.

Next, a special For…Next loop is used—a For Each…Next loop, to be specific. A variable, oTire, is provided. Each time the loop executes, oTire will be set so that it refers to one child of the Tires collection. The first time through the loop, oTire will refer to the first tire on the car; the second time, the second tire; and so forth. Within the loop, a message box displays the current tire's position and condition: "Left Front has 10,000 miles."

Creating Objects

When you execute a CreateObject() function, several things happen under the hood. First, VBScript goes into the registry's HKEY_CLASSES_ROOT hive to look up the class name you specified. That registry will tell VBScript which DLL actually implements that class.

Next, VBScript will load that DLL into memory (if it isn't already loaded into memory). VBScript will assign a reference to the DLL into the variable you specify so that your variable represents the DLL itself.

From then on, the DLL will remain running as long as your script remains running, and you can manipulate the DLL using the variable to which the DLL's reference was assigned. If you want to *release* the DLL early, simply set that variable to Nothing by using Set oCar = Nothing, for example.

AD, files and folders, Windows Management Instrumentation (WMI) are all accessible as COM objects. In fact, there are literally *thousands* of available COM objects—but only about a half-dozen you're likely to find yourself working with at first. One of the important objects is the WSH library, called WScript.

The WScript Object

WScript is the name of a built-in object that is included as part of WSH. WScript provides some useful methods, and is *always available* to your scripts, meaning you don't need to use CreateObject(). For example, try the following script:

```
WScript.Echo "Hello World"
```

This script displays a message box (or outputs to a command line, depending on which script host is executing the script; see the sidebar, "What's a Host?" for more details). Notice that you don't need to use CreateObject() to create a reference to WScript. In fact, CreateObject() is actually a method of the WScript object. The complete, proper syntax for using CreateObject() is:

```
Dim oObject
Set oObject = WScript.CreateObject("object name")
```

So why didn't the previous example use this syntax? Well, you don't have to, really. If you leave out WScript, VBScript figures out what you're trying to do.

What's a Host?

A *script host* is simply an executable that runs on Windows and is capable of processing and executing scripts like the VBScripts you'll write. Microsoft's WSH is the most popular host, or at least the one you'll use most often.

WSH comes in two flavors. The first is implemented at WScript.exe and the second is CScript.exe. Both do pretty much the same thing, and, by default, WScript.exe is the one that runs VBS files when you double-click them. The big difference between the WSH hosts is that WScript is intended for graphical use, and CScript is intended for command-line use.

When you use a statement such as MsgBox or a function such as InputBox(), you'll get the same graphical dialog box regardless of whether you use CScript or WScript. However, when using the intrinsic WScript object's Echo method, for example

```
WScript.Echo "Hello World"
```

you'll get very different results from the two WSH hosts. When WScript executes that method, it displays a dialog box that looks a lot like a message box. When CScript executes it, "Hello World" is output to a command-line.

CScript is useful for writing your own command-line utilities, especially ones you plan to run under Scheduled Tasks. The reason is that the Windows Task Scheduler doesn't provide a graphical environment for tasks, so any message boxes or input boxes will "freeze" the script and prevent it from running properly. By using CScript, you can use WScript.Echo and the script will keep working as a scheduled task.

You can make CScript the default script host, meaning it will execute any VBS files you double-click. Run WScript.exe or CScript.exe for the proper command-line syntax that allows either one to be set as the default.

WScript has two other methods you may use form time to time:

- **WScript.Quit—**This method causes your script to immediately stop executing. You can use this method to provide users with "Cancel" or "Quit" options, if appropriate.
- **WScript.Sleep—**This method provides the number of milliseconds you want your script to pause (for example, WScript.Sleep 1000 to pause for one second), and your script will sit there and wait.

WSH is bundled with some other useful objects. You'll need to use CreateObject() with these:

- **WshController—**This object allows you to instantiate and execute scripts on a remote computer, and get feedback when those scripts finish executing or encounter an error.

📖 Chapter 4 will provide more information about WshController.

- **WScript.Network—**This object provides access to networking capabilities, such as drive and printer mapping.
- **WScript.Shell—**This object provides access to basic Windows Explorer functions, such as creating shortcuts, executing other scripts or applications, working with environment variables, accessing the registry, and more.

> ☞ Take a moment to open the WSH documentation and locate the references for these objects. As this guide doesn't exhaustively cover each and every method and property of these objects, you should review the documentation to see what else they offer. Browsing the documentation in this fashion is the best way to get an idea of what capabilities your scripts can utilize.

Listing 1.4 shows an example of the WScript.Shell object in action.

```
Set oShell = WScript.CreateObject("WScript.Shell")
sDesktopFolder = oShell.SpecialFolders("Desktop")
Set oLink = oShell.CreateShortcut(sDesktopFolder & _
 "\Shortcut.lnk")
oLink.TargetPath = WScript.ScriptFullName
oLink.WindowStyle = 1
oLink.Hotkey = "CTRL+SHIFT+F"
oLink.IconLocation = "notepad.exe, 0"
oLink.Description = "Shortcut Script"
oLink.WorkingDirectory = sDesktopFolder
oLink.Save
Set oLink = Nothing
Set oShell = Nothing
```

Listing 1.4: An example WScriptShell object script.

In Listing 1.4, the script is creating a new WScript.Shell object reference, and letting VBScript get the DLL up and running in memory. This script uses the SpecialFolders() method to retrieve the actual path to the special Desktop folder (often somewhere in C:\Documents and Settings). In addition, the script is using the CreateShortcut() method to create a new shortcut. This method actually creates a new object; in order to set the properties of this new object, the script captures a reference to the object in the oLink variable. Notice that this line of code didn't quite fit on one line of text; the underscore ("_") character tells VBScript that the line of code is continued on the next line of text.

Next, the script set the target for the shortcut to be this actual script, and retrieves the script's path using the WScript object's ScriptFullName property. It then sets several additional properties for the shortcut, and finally saves those settings using the Save method. Notice that both object references are set to Nothing before finishing. This setting isn't necessary; because the script is done, VBScript will clean up after itself. However, it is a good practice to release all object references.

The following example provides a shorter sample script that retrieves useful network information and even maps a network drive. This functionality could be used in a login script.

```
Set oNetwork = WScript.CreateObject("WScript.Network")
WScript.Echo "Domain = " & oNetwork.UserDomain
WScript.Echo "Computer Name = " & oNetwork.ComputerName
WScript.Echo "User Name = " & oNetwork.UserName
oNetwork.MapNetworkDrive "Z:", "\\Server\Share"
```

This example should be easy to follow. It creates a WScript.Network object, then outputs the current user domain, computer name, and user name to the screen. Finally, it maps the Z drive to the UNC \\Server\Share (which must exist, or you'll get an error; feel free to change this to a UNC that exists in your environment).

> ✎ You'll see more of the WScript.Shell and WScript.Network objects in the last section of this chapter, but feel free to spend some time browsing the WSH documentation and experimenting with these two objects. Neither of these objects contains any methods or properties that can permanently damage your OS, computer, or network.

Once you have an idea of what objects are for and how they work, you are ready for an introduction to a real heavy-hitter—FileSystemObject (FSO).

File and Folder Objects

The FSO is included with WSH and is part of the Windows Scripting Runtime. You'll find documentation for the FSO with the rest of the scripting documentation; if you have access to a copy of the MSDN Library (which comes on CD-ROM and DVD), you can find this documentation in the contents at Web Development, Scripting, SDK Documentation, Windows Script Technologies, Script Runtime, FileSystemObject Object.

> ✎ Although it looks strange call something the *FileSystemObject object*, that is correct usage. *FileSystemObject*, all one word, is its name, and you'll commonly see it abbreviated as *FSO*.

The FSO is a powerful object, providing almost complete access to the Windows file system. Notice the word "almost." The FSO doesn't provide any access to NTFS or share permissions, nor does it provide access to file and folder auditing settings.

> 🕮 WMI provides this type of access; Chapter 3 will explore WMI in more detail.

Getting the FSO up and running is easy enough:

```
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
```

> ☞ How do you discover that the official name of the FSO is Scripting.FileSystemObject and not just FileSystemObject? Through experience and by looking at the documentation. Don't be afraid to read the manual when it comes to scripting. You'll save tons of time. At the very least, try to look at someone else's work (for example, check out the samples available at http://www.ScriptingAnswers.com) to see what they did—there's no sense in reinventing the wheel.

Once you have a reference to the FSO (the reference in the previous example is in the oFSO variable), you can start using it. The FSO has three child objects that you should know about:

- **Drive—**This object represents a logical drive on your computer (the FSO doesn't provide access to physical disks, only logical drives)
- **Folder—**This object represents a folder or directory
- **File—**This object represents a single file

See if you can make sense of this example:

```
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
WScript.Echo oFSO.GetDrive("C:").RootFolder.Path
```

The first two lines probably make sense. They're just declaring a variable, then creating the FSO object and assigning a reference to that variable. The third line is a bit thick, though.

You're familiar with WScript.Echo by now; the rest of the line is the FSO's GetDrive() method, which retrieves a reference to a particular logical drive (in this case, the C drive). The result of this method is a Drive object, which has a RootFolder property. Not surprisingly, this property represents the root folder of that drive, and has a Path property that displays the root folder's complete path—C:\ in this example.

The following script works with a folder. Note that you'll need to provide a folder name that actually exists and that you don't mind losing:

```
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
Set oFolder = oFSO.GetFolder("C:\DeleteMe")
oFolder.Delete
```

There is no "Are you sure?" prompt, and the Recycle Bin isn't involved; that folder is history. Oddly, the FSO provides almost two ways to do everything, and here's the other:

```
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
oFSO.DeleteFolder "C:\DeleteMe"
```

See the difference? In the first example, the script used the GetFolder() method to retrieve a Folder object, which represented the folder in which the script is interested. The script then used that object's Delete method to delete it. In the second example, the script used the FSO's direct DeleteFolder method to delete the folder. Same result, slightly different approach.

Files work similarly. Note again that you'll need to provide a filename that exists:

```
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
Set oFile = oFSO.GetFile("C:\MyFile.txt")
oFile.Copy "D:\MyFile.txt"
```

Again, you could shortcut this process by using the FSO's direct CopyFile method, as the following example shows. This example also includes some program logic—you remember the If…Then construct, right?

```
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
If oFSO.FileExists("C:\MyFile.txt") Then
 OFSO.CopyFile "C:\MyFile.txt", "D:\MyFile.txt"
End If
```

Even if the file doesn't exist, your script will run, because the script is using the FSO's FileExists() method to check for the file's existence before trying to copy it. Notice that the If…Then construct doesn't actually seem to have an expression (there is no comparison or equals sign). The reason is that the FileExists() method returns either a True or False; VBScript will execute the interior code if FileExists() returns True, and it will skip the interior code if FileExists() returns False. The following version of this example does exactly the same thing, but explicitly states the condition:

```
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
If oFSO.FileExists("C:\MyFile.txt") = True Then
 OFSO.CopyFile "C:\MyFile.txt", "D:\MyFile.txt"
End If
```

If the D drive doesn't exist, the script will *still* fail with an error message. There is a way around this failure—actually, two ways. The first is

```
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
If oFSO.FileExists("C:\MyFile.txt") Then
 If oFSO.DriveExists("D:") Then
  OFSO.CopyFile "C:\MyFile.txt", "D:\MyFile.txt"
 Else
  WScript.Echo "Can't copy file; D: doesn't exist"
 End If
End If
```

This script uses a nested If…Then construct to catch the possibility that D doesn't exist. This workaround is a good way to handle possible error conditions—anticipate them, and have your script check to see if everything is in place *first.* Another way is to do so is to use

```
Dim oFSO
On Error Resume Next
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
If oFSO.FileExists("C:\MyFile.txt") Then
 OFSO.CopyFile "C:\MyFile.txt", "D:\MyFile.txt"
 If Err.Number <> 0 Then
  WScript.Echo "Error copying file to D: " & Err.Description
 End If
End If
```

This script includes a special statement—On Error Resume Next. This statement tells VBScript to ignore an error if one occurs. VBScript won't stop running the script if an error occurs; instead, it populates a special object named Err with information about the error, and lets you deal with it. You can see the code that has been added to deal with a potential error. If the error number is anything but zero (zero means no error occurred), the error's description is helpfully displayed.

The following script illustrates another FSO trick:

```
Dim oFSO, oTS
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
Set oTS = CreateTextFile("C:\MyOutput.txt")
oTS.WriteLine "Hello, world!"
oTS.Close
WScript.Echo "All done!"
```

This sample uses another FSO child object, called a TextStream object. A TextStream represents a stream of text in a file, or, in plain English, a text file. In this case, the script created a new text file, wrote a line of text to it, then closed the file. You can also use the OpenTextFile() method to open an existing text file, and the ReadLine method to read lines of text from it. In either case, both CreateTextFile() and OpenTextFile() are methods of the FSO that return a TextStream object; that object is assigned to a reference variable (oTS in the previous example). WriteLine and ReadLine are methods of the TextStream object; Close is also a method of the TextStream object and closes the file.

Your First Administrative Script

Your whirlwind introduction to VBScript is almost at a close. Before we dive into anything else, though, let's wrap up everything we've explored so far with a quick walkthrough of a complete script, which you'll write from scratch. Here's the task: Create a text file that contains the name of each user logging onto each computer in your environment.

When you have a task to write a script for, you need to start by breaking down the task into pieces to make it approachable. Otherwise, this task becomes a giant obstacle, and you'll spend 3 weeks on Google looking for a pre-written script. The following list provides a not-necessarily-orderly thought process for this task:

- Write a file—Use the FSO to create a text file, which gets you a TextStream; use that object's WriteLine method to write the content to the file.
- Access the computer and user name—Use the WScript.Network object to get your computer name and user name.
- Store a file on a file server where all users can access the file (you probably suspect that if several users try to open a text file at the same time, they won't all be able to write to it, so you'll need to handle this complication).
- Assign a VBScript as a logon script in AD.

So, the first step is to create a script that is a logon script, and assign it to each user. Have the script open a text file and write out the computer and user names. Potential problem: If two users log on at once (which is probably inevitable), one of them might not be able to write to the file while the other one has the file opened.

Rather than mess around with potential problems, look immediately to find a workaround. Perhaps have each user's logon script write a *separate* file. Computer names are unique, so you can have each script write a file named after the computer. The file can contain the user and computer name. You can write another script to run in the afternoon—after everyone's logged on—that puts all the text files together into one. Maybe not the most elegant solution, but it will get the job done. So the first script looks like this:

```
Dim oFSO, oNetwork, oTS
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
Set oNetwork = WScript.CreateObject("WScript.Network")
Set oTS = oFSO.CreateTextFile("\\Server\Share\" & _
 oNetwork.ComputerName & ".txt", True)
oTS.WriteLine oNetwork.ComputerName & "," & _
 oNetwork.UserName
oTS.Close
```

You can assign this script as a logon script. Most of the work is done on line four: the script asks the FSO to create a text file on \\Server\Share, named after the computer, with a .TXT filename extension. Notice the second parameter, "True." That tells the FSO to overwrite any existing file of the same name—just to be sure. The script then uses the WriteLine method to output the computer name, a comma, and the username, all on one line, to the next file. Finally, the script closes the file. Later, you can run the following script:

```
Dim oFSO, oFile, oFolder, oTS, oTS2
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")
Set oTS = oFSO.CreateTextFile("C:\Names.txt")
Set oFolder = oFSO.GetFolder("\\Server\Share")
For Each oFile In oFolder.Files
 Set oTS2 = oFSO.OpenTextFile("\\Server\Share\" & oFile.Name
 oTS.WriteLine oTS2.ReadLine
 oTS2.Close
Next
oTS.Close
WScript.Echo "Done merging files"
```

The action begins on line four, where the script asks the FSO to get a reference to the folder containing all those text files. It returns a Folder object. One of a Folder object's properties, Files, is actually a collection of individual File objects.

At this point, the script turns to a For Each…Next construct. The construct loops through each child object of the Files collection, representing each one in the variable oFile. The script uses the oFile reference to open each file, write its contents to an output file, then close the file. When it's all over, the script will helpfully display a message telling you that it has finished (otherwise, you might not know when all the files have been processed; scripts don't give any automatic visible indication when they're finished running). Try experimenting with this script on your own and see what you can get it to do.

Summary

Although there is a lot of additional scripting information, much of it isn't of great interest to most administrators. For example, this chapter has completely skipped Dictionary objects, because not many administrators need to use them. Like most other administrators, you're probably more interested in getting up and running than reading about scripting you won't even use.

- If you're really interested in learning more about scripting just for the sake of knowledge, check out the following resources:
- *Managing Windows with VBScript and WMI* (Addison-Wesley)—A no-nonsense guide to scripting for Windows administrators
- *Microsoft Windows 2000 Scripting Guide* (Microsoft Press)—An in-depth guide to everything scripting can do.
- Both are good choices for a scripting administrator's bookshelf, although many of the samples from the *Scripting Guide* are available for free in the Microsoft TechNet Script Center at http://www.microsoft.com/technet/community/scriptcenter/default.mspx.

Things get more fun in the next chapter, in which you'll learn how to write scripts that work with AD, local computer accounts, and more by using the Active Directory Services Interface (ADSI). In Chapter 3, we'll explore scripting a bit more in-dept and start using WMI to query and change computers' configuration information.