realtimepublishers.com™

The Administrator Shortcut Guide™ To

VBScripting for Windows

Don Jones

Introduction to Realtimepublishers

by Don Jones, Series Editor

For several years, now, Realtime has produced dozens and dozens of high-quality books that just happen to be delivered in electronic format—at no cost to you, the reader. We've made this unique publishing model work through the generous support and cooperation of our sponsors, who agree to bear each book's production expenses for the benefit of our readers.

Although we've always offered our publications to you for free, don't think for a moment that quality is anything less than our top priority. My job is to make sure that our books are as good as—and in most cases better than—any printed book that would cost you $40 or more. Our electronic publishing model offers several advantages over printed books: You receive chapters literally as fast as our authors produce them (hence the "realtime" aspect of our model), and we can update chapters to reflect the latest changes in technology.

I want to point out that our books are by no means paid advertisements or white papers. We're an independent publishing company, and an important aspect of my job is to make sure that our authors are free to voice their expertise and opinions without reservation or restriction. We maintain complete editorial control of our publications, and I'm proud that we've produced so many quality books over the past years.

I want to extend an invitation to visit us at http://nexus.realtimepublishers.com, especially if you've received this publication from a friend or colleague. We have a wide variety of additional books on a range of topics, and you're sure to find something that's of interest to you—and it won't cost you a thing. We hope you'll continue to come to Realtime for your educational needs far into the future.

Until then, enjoy.

Don Jones

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://msdn2.microsoft.com/en-us/library/d1wf56tt.aspx.

> 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?"
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.

Chapter 2: Working with ADSI

With all the hype about WMI, ADSI has been getting short shrift, which is a shame because ADSI is such a useful tool. A possible reason that ADSI is overlooked by most administrators is that it is poorly named. Many administrators assume that a tool named Active Directory Services Interface deals solely with AD. The truth is that ADSI *can* work with AD, but offers many additional capabilities.

ADSI Without a Directory

Some of ADSI's most useful features have nothing to do with *any* directory, let alone AD. Listing 2.1 shows a favorite sample scripts; this script should work without modification in any environment. It contacts any Windows file server—domain member or not—and tells you which users currently have a particular shared file open.

```
‘ first, get the server name we want to work with
sServerName = InputBox ("Server name to check")

‘ get the local path of the file to check
sFilename= InputBox ("Full path and filename of the " & _
 "file on the " & _
 "server (use the local path as if you were " & _
 "at the server console)")

‘ bind to the server’s file service
Set oFileService = GetObject("WinNT://" & sServerName & _
 "/lanmanserver,fileservice")

‘ scan through the open resources until we
‘ locate the file we want
bFoundNone = True

‘ use a FOR...EACH loop to walk through the
‘ open resources
For Each oResource In oFileService.Resources

 ‘ does this resource match the one we’re looking for?
On Error Resume Next
 If oResource.Path = sFilename Then
 If Err <> 0 Then
  WScript.Echo "Couldn’t locate the resource, or " & _
   "permission denied."
 End If
  ‘ we found the file - show who’s got it
  bFoundNone = False
  WScript.Echo oResource.Path & " is opened by " & oResource.User
 End If
Next

‘ if we didn’t find the file open, display a msg
If bFoundNone = True Then
 WScript.Echo "Didn’t find that file opened by anyone."
End If
```

Listing 2.1: Sample ADSI script that enables you to determine which users have a shared file open.

This script provides a great way to learn how ADSI works outside of directories. The script starts off by asking you for a server name to check. This name should be entered without backslashes. Next, you must type the path and filename of the open file you're curious about. This path needs to be the local path on the server. For example, suppose you have a file server that contains a folder named D:\Shares\Sales. This folder is shared as SalesData, and it contains an Excel spreadsheet named Forecasts.xls. Users access this file at \\Server\SalesData\Forecasts.xls. However, you would provide this script with D:\Shares\Sales\Forecasts.xls, because that's the local path as far as the server is concerned.

Next, the script begins to use ADSI and uses the Set statement. In the previous chapter, I explained that the Set statement is used to assign an object reference to a variable. That's exactly what's going on here. However, in the previous chapter, I also told you that the CreateObject() function was used to load DLLs into memory and obtain that reference—this script is using GetObject() instead.

In the case of ADSI, you don't *want* to load a DLL into memory. You want to reach out and connect to some service that is already running on a remote machine. Thus, rather than *creating* an object, you just want to *access* an existing, remote object. Hence, GetObject(). The argument passed to GetObject() tells VBScript what to go get. In this case, the WinNT:// name tells VBScript that you're using ADSI's Windows NT directory provider. The script also passes the server name you entered, then specifies a connection to the lanmanserver object on that server. Specifically, it wants a fileservice named lanmanserver.

So, we've established that ADSI isn't just for AD. ADSI is actually a generic directory access technology. Even NT boxes expose many of their services in a directory-like format that is accessible through ADSI's WinNT provider. Let's take a closer look at the ADSI call in Listing 2.1:

```
' bind to the server's file service
Set oFileService = GetObject("WinNT://" & sServerName & _
 "/lanmanserver,fileservice")
```

In this call:

- The script uses the Set keyword because it is assigning an object reference to a variable.
- The script specifies the variable—in this case, oFileService.
- The script specifies the GetObject() function, which has nothing specifically to do with ADSI; it is simply a generic function that connects to the object you specify. Generally, this function connects to objects that are actually services running on the local machine or a remote machine.
- The script specifies the ADSI provider—in this example, WinNT://.

The provider names are case-sensitive; thus, in this case, winnt:// will *not* work.

- Next is the server name, which can be the local box or a remote one, it doesn't matter. In this example, the script uses a variable to store the server name, and appends it into the ADSI call by using VBScript's string-concatenation operator, the ampersand (&).
- Next, the script names an object or resource on the computer, which is the object or resource to which ADSI will actually connect.

> 🕮 Later in this chapter, we'll explore user and group names as the object to connect with; in the example that Listing 2.1 shows, the object or resource is lanmanserver, which happens to be the internal name of Windows' Server service.

- Optionally, you can specify an object class. In this example, the script specifies fileservice. If you don't specify an object class, ADSI will latch onto the first object or resource it finds that matches the name specified. For example, if I had a user named lanmanserver, then ADSI might grab that user instead of the Server service. However, by specifying the object class, I ensure that ADSI will grab the resource in which I'm interested.

After the first 11 lines of the script, it gets less complicated. On line 15, the script assigns the value True to a variable named bFoundNone, which is a sort of reminder that I haven't yet found any users who were using the specified file.

Keep in mind that, at this point, oFileService essentially represents the Server service on whatever server the script specifies. The Server service exposes a list of every resource it is currently managing. Thus, on line 19, I walk through each one of those resources in a For Each…Next loop. Each time through the loop, oResource will represent the current resource.

On line 22, the script performs the actual comparison: Does the current resource's Path property equal the filename that you typed into the script? If so, the script sets bFoundNone to the value False, because it has clearly found someone who is using the file. Next, the script outputs the resource's path and the user name that has the file open. *The script doesn't stop the script at this point*. The reason is that than one user can have a file open, and I want to find all of them. Thus, the script has the For Each…Next loop continue through the whole *collection* of resources.

When the loop is done, the script displays a message if it hasn't found any users who have the file open. If the script didn't do so and nobody had the specified file open, the script would end with no visual indication, leaving you to wonder whether the script was still running.

How This Script Was Designed

I have been working with Windows since the NT 3.1 days. Until Win2K, administrators used a management application called Server Manager. With it, you could double-click a server and get a list of every resource that the server was currently managing—namely, open shared files. To discover which user had a file open, you simply opened this list—which on a busy file server would have thousands of entries—and start scrolling through it, looking for the filename in which you were interested. Of course, by the time you got to the end of the list, it was out of date and you would have to close the list, reopen it, and start over. However, it never took more than an hour or so to go through the list.

The script that Listing 2.1 shows performs the same task as the Server Manager utility—it simply does so faster and with an infinitely better attention span to its assigned task. In designing this script, I simply wanted to duplicate—and speed—the task I already knew how to do manually. With a little research, I discovered that ADSI could perform this task for me.

ADSI Providers

Now that we've explored a little about how ADSI works, it's probably time for a more formal introduction. ADSI is Microsoft's technology for generic directory access; the fact that ADSI has Active Directory in the name does *not* mean that it works only with AD. ADSI is preinstalled on Win2K and later, and its core components are also installed with the Directory Services client for NT. ADSI ships with three key providers:

- WinNT—Provides access to NT domains, local computer resources (including Win2K and later boxes), and local security accounts (for Win2K and later).
- Lightweight Directory Access Protocol (LDAP)—Enables you to connect to any LDAP-enabled directory, such as Lotus Notes/Domino, Novell Directory Services (NDS), Microsoft Exchange Server 5.0 or 5.5, and Microsoft AD.

There is also a specific NDS provider, but LDAP is just as useful with the latest versions of NDS.

- NWCOMPAT—Provides connectivity to Novell NetWare 2.x and 3.x binderies (should you happen to work in the Smithsonian where they surely have some of these relics on display).

All of the providers work slightly differently. And let me emphasize once again that these are *case-sensitive*. In other words, ldap is *not* the same thing as LDAP, and ADSI will give vague errors if you use the incorrect provider name.

The WinNT Provider

You've already seen the basic syntax for the WinNT provider:

```
WinNT://host/object,class
```

Host in this case can be a server, workstation, or domain name. If you provide a domain name, your client will follow all the usual rules for locating a domain controller (for example, querying WINS, querying DNS, and so on). Specifying a domain name is useful if you want to mess around with domain usernames and groups; you can also specify the name of a domain controller, of course, rather than the domain name. It's your choice. If you perform an operation that requires an object to be changed (such as changing a user's password), then ADSI will automatically reconnect to the Primary Domain Controller (PDC), the only domain controller in an NT domain that can make changes to the domain (remember that Backup Domain Controllers—BDCs—are read-only).

Pay close attention: *You can connect to AD by using the WinNT provider*. The reason is that AD *emulates* an NT domain. There is a PDC Emulator that runs on one domain controller in every AD domain, and every AD domain controller can provide services to downlevel, NT-based clients.

The LDAP Provider

NT domains are a flat namespace, which is a fancy way of saying that there are no organizational units (OUs), containers, sites, and so forth. Thus, if you're connecting to AD via the WinNT provider, you can't modify OUs, containers, sites, and the things that NT domains don't have. To use the full power of AD through ADSI, you must use the slightly more complex LDAP provider, which works like this:

```
LDAP://fqdn
```

Not very exciting, is it? Here's an actual example:

```
LDAP://cn=donj,ou=research,dc=scriptinganswers,dc=com
```

You have to use LDAP-style naming to provide the *fully-qualified domain name* (FQDN) of the object to which you want to connect. There is no need to specify object classes because you're being very specific: the example shows a user named donj, in an OU named research, in a domain named scriptinganswers.com. The order of these components is important. You must start with the object name, then the OU that contains the object, any parent OUs, then the domain name components from left to right (for example, scriptinganswers first, then com second, for scriptinganswers.com).

When you plug this query into GetObject(), the output depends on the query. If you query a user, the output will be a user object. If you query a computer, you'll get a computer object back. However, you can't query non-directory items, such as lanmanserver; although the WinNT provider has access to a number of non-directory items, LDAP can only access what is in the directory. What you can do with the object that GetObject() returns depends on what type of object it is. For example, with a user object, you can:

```
Set oUser = GetObject _
("LDAP://cn=donj,ou=research,dc=scriptinganswers,dc=com")
oUser.ChangePassword "password", "BetterPassword!"
```

In this example, the User object, as you can see, sports a ChangePassword method that can be used to change the password. You must know the old password in order to change it to a new password.

If you query a Group object, you can change the group's type:

```
Const ADS_GROUP_TYPE_GLOBAL_GROUP = &h2
Const ADS_GROUP_TYPE_LOCAL_GROUP = &h4
Const ADS_GROUP_TYPE_UNIVERSAL_GROUP = &h8
Const ADS_GROUP_TYPE_SECURITY_ENABLED = &h80000000

Set oGroup = GetObject _
   ("LDAP://cn=Writers,dc=scriptinganswers,dc=com")
oGroup.Put "groupType", _
  ADS_GROUP_TYPE_GLOBAL_GROUP + ADS_GROUP_TYPE_SECURITY_ENABLED

oGroup.SetInfo
```

In this example, the script first defines four constants. Constants, as you may remember, are a way of assigning a friendly name to a difficult-to-remember value. They also make your script a bit easier to read; in this case, the names are easier to figure out than the hexadecimal values that the names represent.

Next, the script uses an LDAP call to ADSI to retrieve a group named Writers. The script then uses the Put method of the Group object. Most objects returned by GetObject()from an LDAP query will support the Put and Get methods. These methods allow you to write and read, respectively, any property of the object. In the previous example, the script is writing the groupType property, which, not surprisingly, tells AD what type of group this is.

After specifying the property name, the script needs to specify the value to be put into the property. All you need to do is specify the correct constant name, and if you want the group to be a security group (as opposed to a distribution group), add that constant into the mix. In this example, the group is a global security group.

Whenever you use Put to make changes to an object's properties (or *attributes*), you must finish off with the SetInfo method to save your changes. The reason is that ADSI caches your changes as you make them, but doesn't actually send them off to the domain controller until you execute SetInfo. This technique helps make ADSI a bit more efficient.

An ADSI Shortcut

The Microsoft Scripting Guys—Microsoft employees who promote scripting in their spare time, and who are also full-time members of the Windows product team—have created several labor-saving devices for scripters. One of their nifty tools is the ADSI Scriptomatic, which you can download for free from http://www.microsoft.com/technet/community/scriptcenter/tools/admatic.mspx. It's an HTML Application (HTA), which means you'll need to have Internet Explorer (IE) installed to run it. Figure 2.1 shows the ADSI Scriptomatic.

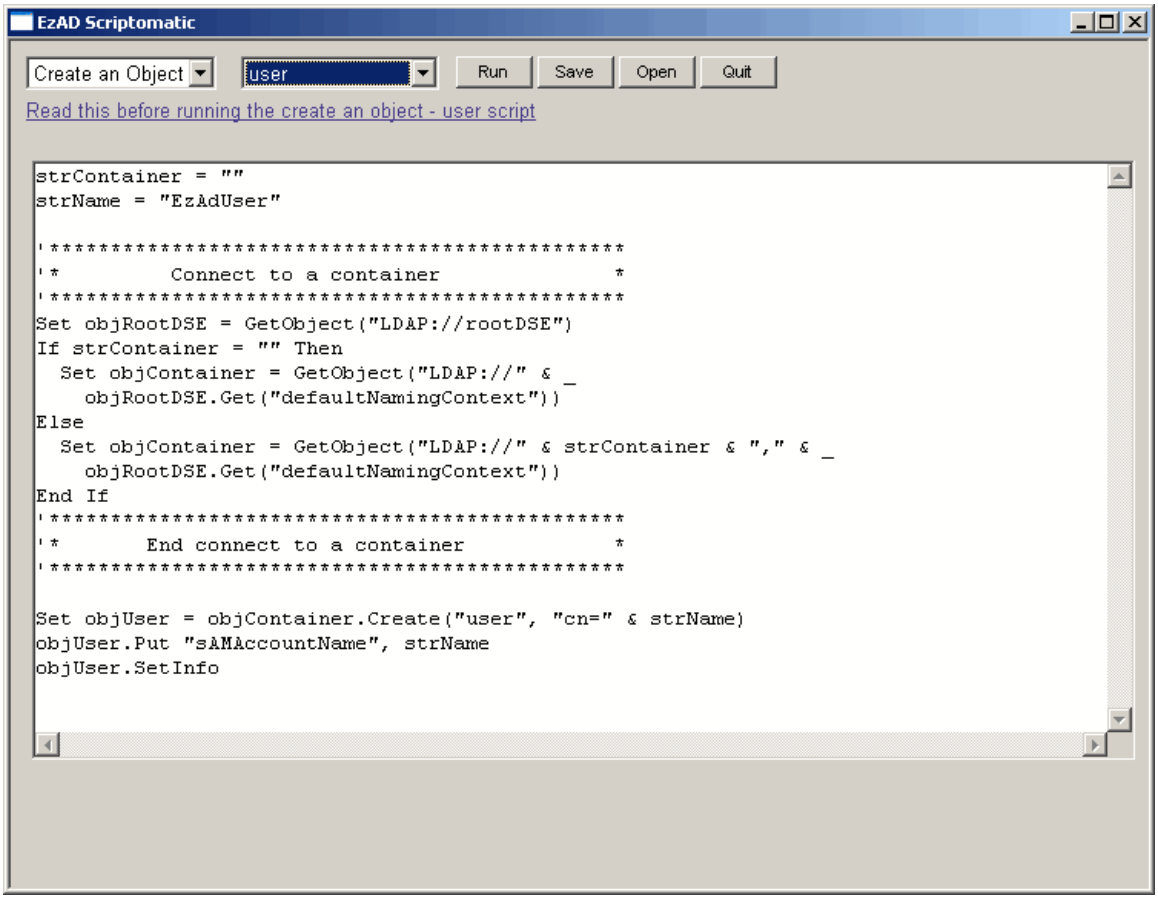

Figure 2.1: The ADSI Scriptomatic generates ADSI scripts for you.

Basically, you just tell this tool what you want to do—Create an Object, in this case—and what object class you want to use. I selected the User class, so the tool generated a script to create a user. Let me briefly walk through this script, because it's the same basic template used for all of the tool's scripts. Understanding it will help you better understand how ADSI works.

The first two lines simply set up two string variables:

```
strContainer = ""
strName = "EzAdUser"
```

These contain the name of the container in which the new object should be created, and the name that the new object should be given. You would obviously change these to the appropriate names for your environment.

Notice that the script uses a variable naming convention; per best practice, the tool uses a three-letter prefix as specified in Hungarian notation, which is a Microsoft-favored notation for VBScript.

Next, as the comment says, the script connects to the container. First, it connects to the root of AD, for whichever domain the computer running the script is a member:

```
'*********************************************
'*         Connect to a container              *
'*********************************************
Set objRootDSE = GetObject("LDAP://rootDSE")
```

rootDSE is a special LDAP object that exists on all LDAP servers. Any LDAP server (version 3 or later) will respond correctly to a query for the rootDSE.

If a container name has not been specified, the script connects to the default naming context of the domain's root. Otherwise, the script connects to the default naming context of the specified container. Either way, what you get back is a container of some kind—the default Users container or a specified OU:

```
If strContainer = "" Then
  Set objContainer = GetObject("LDAP://" & _
    objRootDSE.Get("defaultNamingContext"))
Else
  Set objContainer = GetObject("LDAP://" & strContainer & "," & _
    objRootDSE.Get("defaultNamingContext"))
End If
'*********************************************
'*       End connect to a container            *
'*********************************************
```

Next, the script uses the container's Create method to create an object. The script specifies the class of the object as user and specifies the new user's common name, which is cn= plus whatever is in the variable strName. The Create method returns a reference to the newly created object, which is stored in the variable objUser.

```
Set objUser = objContainer.Create("user", "cn=" & strName)
```

Next, the script uses the new object's Put method to set the sAMAccountName attribute to the new object's username. SetInfo is called to save everything, and the script is finished.

```
objUser.Put "sAMAccountName", strName
objUser.SetInfo
```

sAMAccountName might look like a case of the Shift key gone wrong, but it's actually correct. SAM stands for Security Accounts Manager; the attribute *should* be named "SAMAccountName," but AD's schema naming standards specify a lowercase letter for the first letter in the attribute name, so sAMAccountName it is.

Querying Global Catalog Servers

In an AD domain, one or more servers fill the role of Global Catalog (GC) server, a server that holds information about every object in the entire forest. You can query GCs directly by using ADSI. The following example uses the special GC: provider to get a list of GC servers' domains:

```
Set oList = GetObject("GC:")
For Each oServer in oList
 WScript.Echo oServer.ADsPath
Next
```

Of course, once you have a GC's name (or ADsPath), you can query the server directly to retrieve information.

Microsoft provides a sample script at http://support.microsoft.com/?kbid=252490 that will tell you whether a particular user principle name (UPN) exists in a forest so that you can avoid making duplicate user objects.

Useful ADSI Scripts

At this point, you should have some idea of how ADSI works and how you can use it within your scripts. The best way to learn more details is to actually explore useful ADSI scripts, so I'll give you a bunch of short little scriptlets and explain how they work. Most of these are scriptlets that you would incorporate into your own, larger scripts; they're not necessarily intended to be run entirely on their own. Pay special attention to where I'm using the WinNT and LDAP providers; you'll get a better feel for what capabilities each can provide to your scripts.

User Account Scripts

The ability to batch create users is a great reason to write a script. Suppose you have a text file (or an Excel spreadsheet) that lists the users you need to create. From Excel, save the file as a CSV and use the script that Listing 2.2 shows.

```
' PART 1: Open up the text file
Dim oFSO, oTS
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTS = oFSO.OpenTextFile("c:\users.csv")

'let's agree that the CSV file's first line will
'be the column names, and that they will be:
' Username, Fullname, Description, HomeDir

' PART 2: Get a reference to the
' Windows NT or AD domain using ADSI
Dim oDomain
Set oDomain = GetObject("WinNT://NT4PDC")


' PART 3: Open an output text file
' to store users' initial passwords
Dim oTSOut
Set oTSOut = oFSO.CreateTextFile("c:\passwords.csv",True)


' create the necessary variables
Dim sUserID, sFullName, sDescription
Dim sPassword, sHomeDir

'skip the first line of the file
Dim sLine, sData
sLine = oTS.ReadLine

' now go through the file one
' line at a time
Do Until oTS.AtEndOfStream

 'read the line and split it
 'on the commas
 sLine = oTS.ReadLine
 sData = Split(sLine,",")

 ' get the user information from this row
```

```
 sUserID = sData(0)
 sFullName = sData(1)
 sDescription = sData(2)
 sHomeDir = sData(3)

 ‘ make up a new password
 sPassword = Left(sUserID,2) & _
  DatePart("n",Time) & _
  DatePart("y",Date) & _
  DatePart("s",Time)

 ‘ create the user account
 Set oUserAcct = oDomain.Create("user",sUserID)
 oUser.Put "sAMAccountName", sUserID

 ‘ set account properties
 oUserAcct.SetPassword sPassword
 oUserAcct.FullName = sFullName
 oUserAcct.Description = sDescription
 oUserAcct.HomeDirectory = sHomeDir

 ‘ save the account
 oUserAcct.SetInfo

 ‘ write password to file
 oTSOut.Write sUserID & "," & sPassword & VbCrLf

Loop


‘ PART 6: Final clean up, close down
oRS.Close
oTS.Close
WScript.Echo "Passwords have been written to c:\passwords.csv."
```

Listing 2.2: Script to batch create users.

> 🕮 This example script employs fancy stuff with text files that I haven’t yet introduced; we’ll explore them a bit later in this chapter.

The script that Listing 2.2 shows assumes that your column names are Username, Fullname, Description, and HomeDir, and that the first row of your spreadsheet (or CSV file) uses those column names. Because it utilizes the WinNT provider, this script will work with any Windows domain; in an AD domain, users will be created in the default Users container, and you can move them from there. The script makes up a non-random password for each user, which it writes to a text file for your reference.

Group Scripts

One common task that you might want to perform for a group is to determine a group's membership, especially with security-sensitive groups such as the local Administrators group. In this case, the WinNT provider can be helpful, even in AD domains, because the WinNT provider can work with local groups on member and standalone computers. The following script provides a list of every local group on a designated computer and every member of each group:

```
Dim sComputer, cGroups, oGroup, oUser
sComputer = "ClientA"
Set cGroups = GetObject("WinNT://" & sComputer & "")
cGroups.Filter = Array("group")
For Each oGroup in cGroups
 WScript.Echo oGroup.Name & " members:"
 For Each oUser in oGroup.Members
  WScript.Echo oUser.Name
 Next
Next
```

This script connects to the remote computer and retrieves every object that the computer's WinNT provider can handle. It then adds a one-element array containing the string "group" to the resulting collection's Filter property. Doing so has the effect of filtering the collection to only include (or at least *seem* to include) objects of the class "group." Then the script uses two For Each…Next loops. The outer loop enumerates the groups on the computer, and the inner loop enumerates the members of each group.

If you want to add a user to a group, you can use a script similar to the following example. This script uses ADSI to add a user to an AD group:

```
Const ADS_PROPERTY_APPEND = 3

Set objGroup = GetObject _
    ("LDAP://cn=Special,cn=Users,dc=scriptinganswers,dc=com")

objGroup.PutEx ADS_PROPERTY_APPEND, "member", _
 Array("cn=donj,ou=Executives,dc=scriptinganswers,dc=com")

objGroup.SetInfo
```

This script starts by getting a reference to the group that is to be modified. Keep in mind that the members of a group are part of the *group* object, not the user. In other words, user objects don't contain a list of the groups to which the user belongs. Group objects contain lists of users (and other groups), not the other way around. Thus, if you want to modify a group's membership, you must access the group object.

Next, the script modifies the group's "member" property. Rather than complete overwrite this property, the script simply seeks to append a value to the property. The special PutEx method allows you to pass the property a value, which is stored in a constant, specifying that the script is only appending data to the property, not completely rewriting the property.

The script then provides the property in question—"member"—and the value to be appended. PutEx takes arrays, so the script uses the VBScript Array() function. This function accepts a list of values (just one value, in this example, but you could provide a whole comma-delimited list of values) and turns the list into an array that PutEx can deal with. The value that the script provides is the FQDN of the user to be added to the group (you can also specify a group to add to the group). Finally, the script uses SetInfo to write everything back to the domain controller and saves the change.

Computer Account Scripts

AD does amazingly cool stuff when a Win2K or later machines logs on. The NetLogon service actually feeds some basic inventory data about the machine into AD—kind of a mini-Systems Management Server (SMS). Some of this information is displayed in the computer's Properties dialog box in the Microsoft Management Console (MMC) Active Directory Users and Computers console; you can also query this information from AD by using a script similar to the example that Listing 2.3 shows.

```
strContainer = "ou=Domain Controllers"
strName = "BRAINCORENET"

On Error Resume Next

'*********************************************
'*          Connect to an object                *
'*********************************************
Set objRootDSE = GetObject("LDAP://rootDSE")
If strContainer = "" Then
  Set objItem = GetObject("LDAP://" & _
    objRootDSE.Get("defaultNamingContext"))
Else
  Set objItem = GetObject("LDAP://cn=" & strName & "," & _
    strContainer & "," & _
    objRootDSE.Get("defaultNamingContext"))
End If
'*********************************************
'*         End connect to an object            *
'*********************************************

WScript.Echo "   DNS: " & objItem.Get("dnsHostName")
WScript.Echo "    OS: " & objItem.Get("operatingSystem")
WScript.Echo "OS ver: " & objItem.Get("operatingSystemVersion")
WScript.Echo "    SP: " & _
 objItem.Get("operatingSystemServicePack")
WScript.Echo "Hotfix: " & objItem.Get("operatingSystemHotfix")
```

Listing 2.3: An example script to query basic inventory information about a machine.

As Listing 2.3 shows, I used the ADSI Scriptomatic as the starting point for my script, and filled in the OU and computer name that I was interested in. The Scriptomatic code connects to the computer object in the directory; the script then uses the Get method to retrieve a few properties. Try this with Win2K machines, which are interesting because they have service packs and other items installed. Querying a Windows 2003 server gets you the following output:

```
        DNS: braincorenet.braincore.pri
         OS: Windows Server 2003
     OS ver: 5.2 (3790)
```

Notice that the last two properties aren't displayed because I haven't applied any service packs or anything to this computer.

Where Does He Get This Stuff?

You might be wondering where I discovered the "operating SystemServicePack" attribute and some of the other properties. Knowing what is available to you is a big part of scripting. In the previous chapter, I recommended becoming friendly with the VBScript documentation because browsing would help you determine what is possible with VBScript. I recommend the same thing for ADSI—get to know your objects.

You can do so by browsing the Internet. Plenty of sites, including Microsoft's and my own ScriptingAnswers.com, provide oodles of examples to learn from. The easiest way to get to know ADSI, however, is built right into Windows itself. Open a command line on a domain controller. Change to the \Windows\System32 folder. Run Regsvr32 schmmgmt.dll, then run Mmc. From the Console menu, select Add/Remove Snap-Ins, and add the Active Directory Schema snap-in (the business about running Regsvr32 is necessary because the Schema snap-in isn't registered and available by default; thus, you only need to do it the first time).

Next, click on the "computer" class in the left tree view. In the right pane, you see a list of every possible attribute for a computer object; you can do the same for any AD object, including users, groups, and more. Browsing through the list reveals plenty of potentially useful properties, all of which are accessible from your scripts.

Computer Management Scripts

The WinNT provider is very useful for working with local computer accounts. For example, one security task that often goes undone is a regular password change for the local Administrator account on each of your computers. The script that Listing 2.4 shows will read computer names (one name per line) from a text file named c:\computers.txt, then change the Administrator account password on each of them. This script will skip over any computers that the script can't contact at the time, writing their names to c:\skipped.txt. You can then rename c:\skipped.txt to c:\computers.txt and have the script retry those computers later.

```
‘Create a FileSystemObject
Set oFS = CreateObject("Scripting.FileSystemObject")

‘Open a text file of computer names
‘with one computer name per line
Set oTS = oFS.OpenTextFile("c:\computers.txt")
Set oTSOut = oFS.CreateTextFile("c:\skipped.txt")

‘go through the text file
Do Until oTS.AtEndOfStream

 ‘get the next computer name
 sComputer = oTS.ReadLine

 ‘retrieve that computer’s local Admin
 On Error Resume Next
 Set oUser = GetObject("WinNT://" & _
  sComputer & "/Administrator,user")

 If Err = 0 Then
   ‘reset the password
   oUser.SetPassword "New!Ad3in"

   ‘save the change
   oUser.SetInfo
 Else
   oTSOut.WriteLine sComputer
 End If
 On Error Goto 0

Loop

‘close the text file
oTS.Close
oTSOut.Close
MsgBox "Done!"
```

Listing 2.4: Example script to change the password for the local Administrator account on each of your computers.

You can see the FileSystemObject object in action; you learned about that object in the previous chapter. As you can see, the script reads through an entire text file, which is represented by the variable oTS. The object type for that is called a TextStream, because a text file is really just a big stream of text characters. One of the properties of a TextStream object is AtEndOfStream, which is set to the value True when the end of the file is reached. Thus, the script’s Do Until loop will continue reading the text file until the end of the file is reached.

The ReadLine method of the TextStream is used to read a line of text and store it in the variable sComputer. That variable, and ADSI’s WinNT provider, are used to connect to the specified computer’s Administrator account. What comes back is a User object, and the script uses its SetPassword method to set a new password. The script then calls SetInfo just to be sure everything is saved, although technically with SetPassword, SetInfo isn’t necessary.

Notice the On Error Resume Next object. This object tells VBScript to keep running the script even if it encounters an error, which prevents VBScript from stopping the script if it tries to connect to a computer that is currently turned off. Right after it tries to connect, the script checks the value of the special built-in Err object. If the value of this object is zero, no error occurred and the script can perform the password change. If Err is something other than zero, an error occurred, and the script writes the offending computer's name out to a second TextStream, which is c:\skipped.txt. On Error Goto 0 tells VBScript to start paying attention to errors again.

Scripting Batch Operations

A *batch operation* is any operation that needs to run over and over. For example, creating one hundred users in a script is a good example of a batch operation, and is certainly a task you would rather script than perform manually. The first example I showed you, in which you use a script to determine which users have a shared file open, is another batch operation.

On my Web site at http://www.ScriptingAnswers.com, I provide several scripts that perform batch ADSI operations. One of the most useful is a generic script template that can be used to perform some operation against every user and/or computer account in a domain (see Listing 2.5).

```
'connect to the root of AD
Dim rootDSE, domainObject
Set rootDSE=GetObject("LDAP://RootDSE")
domainContainer = rootDSE.Get("defaultNamingContext")
Set oDomain = GetObject("LDAP://" & domainContainer)

'start with the domain root
WorkWithObject(oDomain)

Sub WorkWithObject(oContainer)
 Dim oADObject
 For Each oADObject in oContainer
  Select Case oADObject.Class
   Case "user"
    'oADObject represents a USER object;
    'do something with it
    '** YOUR CODE HERE**
   Case "computer"
    'oADObject represents a COMPUTER object;
    'do something with it
    '** YOUR CODE HERE**
   Case "organizationalUnit" , "container"
    'oADObject is an OU or container...
    'go through its objects
    WorkWithObject(oADObject)
  End select
 Next
End Sub
```

Listing 2.5: A generic script template that you can use to perform an operation against every user and/or computer account in a domain.

You can see where in Listing 2.5 you would insert your own code. You will use the variable oADObject, which represents either a computer object or a user object. The current OU is always represented by oContainer, and this script will iterate through each and every OU in the domain and find every computer or user within.

For example, suppose your company moved its offices, and you wanted to write a script that changed every user account's ZIP code to 98053. You could use the template script with just a couple of additional lines of code, as Listing 2.6 shows.

```
'connect to the root of AD
Dim rootDSE, domainObject
Set rootDSE=GetObject("LDAP://RootDSE")
domainContainer = rootDSE.Get("defaultNamingContext")
Set oDomain = GetObject("LDAP://" & domainContainer)

'start with the domain root
WorkWithObject(oDomain)

Sub WorkWithObject(oContainer)
 Dim oADObject
 For Each oADObject in oContainer
  Select Case oADObject.Class
   Case "user"
    'oADObject represents a USER object;
    'do something with it
    oADObject.Put "postalCode", "98053"
    oADObject.SetInfo
   Case "computer"
    'oADObject represents a COMPUTER object;
    'do something with it
    '** YOUR CODE HERE**
   Case "organizationalUnit" , "container"
    'oADObject is an OU or container...
    'go through its objects
    WorkWithObject(oADObject)
  End select
 Next
End Sub
```

Listing 2.6: Using the generic template as a starting point for a script to change users' zip code.

I've boldfaced the two additional lines. Notice that I didn't add any code to the section that deals with computer accounts, meaning computers won't be changed—only users.

Another example is to create a text file listing each user and the user's OU. Combining what you learned about the FileSystemObject (in the previous chapter), you might modify the template script as Listing 2.7 shows.

```
'connect to the root of AD
Dim rootDSE, domainObject
Set rootDSE=GetObject("LDAP://RootDSE")
domainContainer = rootDSE.Get("defaultNamingContext")
Set oDomain = GetObject("LDAP://" & domainContainer)

'create a file
Dim oFSO, oTS
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTS = oFSO.CreateTextFile("c:\output.txt",True)

'start with the domain root
WorkWithObject(oDomain)

oTS.Close
MsgBox "All done, boss."

Sub WorkWithObject(oContainer)
 Dim oADObject
 For Each oADObject in oContainer
  Select Case oADObject.Class
   Case "user"
    'oADObject represents a USER object;
    'do something with it
    oTS.WriteLine oADObject.Get("distinguishedName")
   Case "computer"
    'oADObject represents a COMPUTER object;
    'do something with it
    '** YOUR CODE HERE**
   Case "organizationalUnit" , "container"
    'oADObject is an OU or container...
    'go through its objects
    WorkWithObject(oADObject)
  End select
 Next
End Sub
```

Listing 2.7: Using the generic template as a starting point to list each user and user's OU.

This script will write each user's FQDN to a text file named c:\output.txt. Hopefully, this batch template will prove useful in helping you write your own batch ADSI scripts.

Summary

In this chapter, I've introduced you to ADSI. You learned about two key providers, WinNT and LDAP, and how to use them. In addition, we explored how to create new directory objects, connect to services on remote machines, and more. ADSI is a large part of the two-part approach to improved Windows administration through scripting. ADSI provides remote administration capabilities, batch processing capabilities, and more. The other half is WMI, which I'll cover in the next chapter.

Chapter 3: Working with WMI

Windows Management Instrumentation (WMI) is the most talked-about technology for managing Windows computers since… well, since TCP/IP, probably. Unfortunately, WMI comes across as the Windows equivalent to quantum physics—it seems complex, nobody's really explaining it clearly, and the documentation requires a few degrees to understand.

The goal of this chapter is to clear some of the fog. It will steer clear from the technically deep, murky world of WMI and stick with the fun and easy-to-understand aspects of WMI. You'll learn enough to use WMI in administrative scripts, which is most likely what you care about most.

Classes and Queries

Everything in WMI starts with *classes*. A class represents some bit of computer hardware, the OS, an application, or something. A class is really an abstract description. For example, there is a class called Win32_LogicalDisk, which describes a logical disk. It has properties such as BlockSize, DriveType, FreeSpace, and FileSystem, which are all things you would naturally associate with logical disks in Windows.

Each logical disk actually installed under Windows is referred to as an *instance* of the Win32_LogicalDisk class. If you have a C drive and a D drive, that is two *instances* of the class. Instances represent actual, living occurrences of the thing that a class describes. Although classes are interesting, you will most often want to deal with the instances. Consider the following example:

1. On a Windows XP machine, select the Start menu, click Run, type

   ```
   wbemtest
   ```

   and click OK. You should see the dialog box that Figure 3.1 shows.

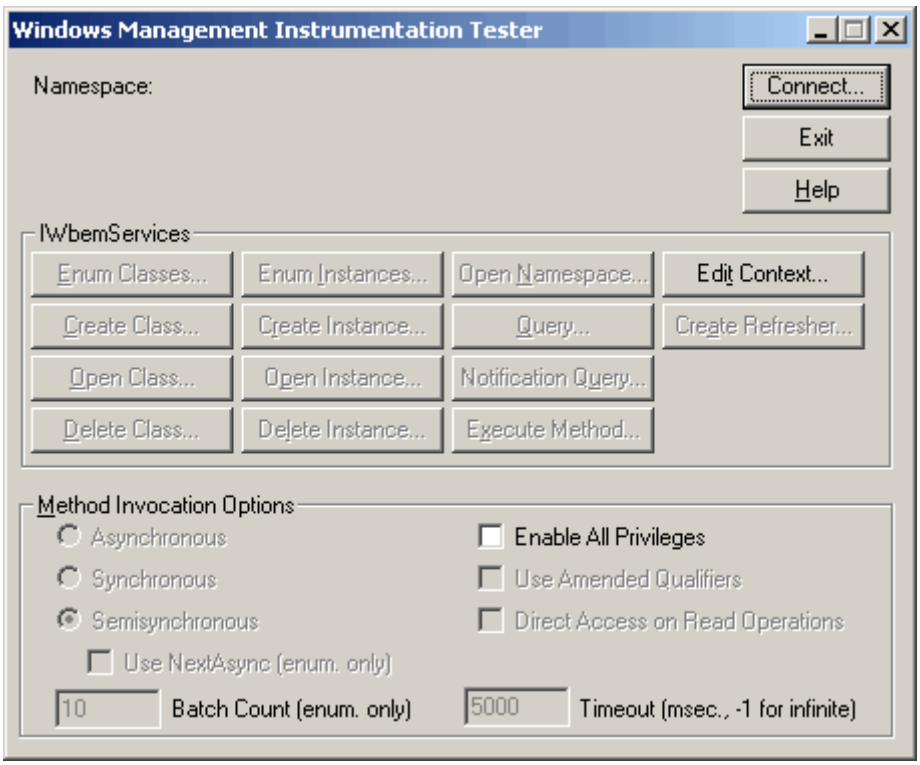

Figure 3.1: The WBEMTEST tool.

2. Click Connect. In the Namespace box, type

   `root\cimv2`

   as Figure 3.2 shows. This namespace is the basic WMI namespace that you'll be using most of the time. Click Connect.

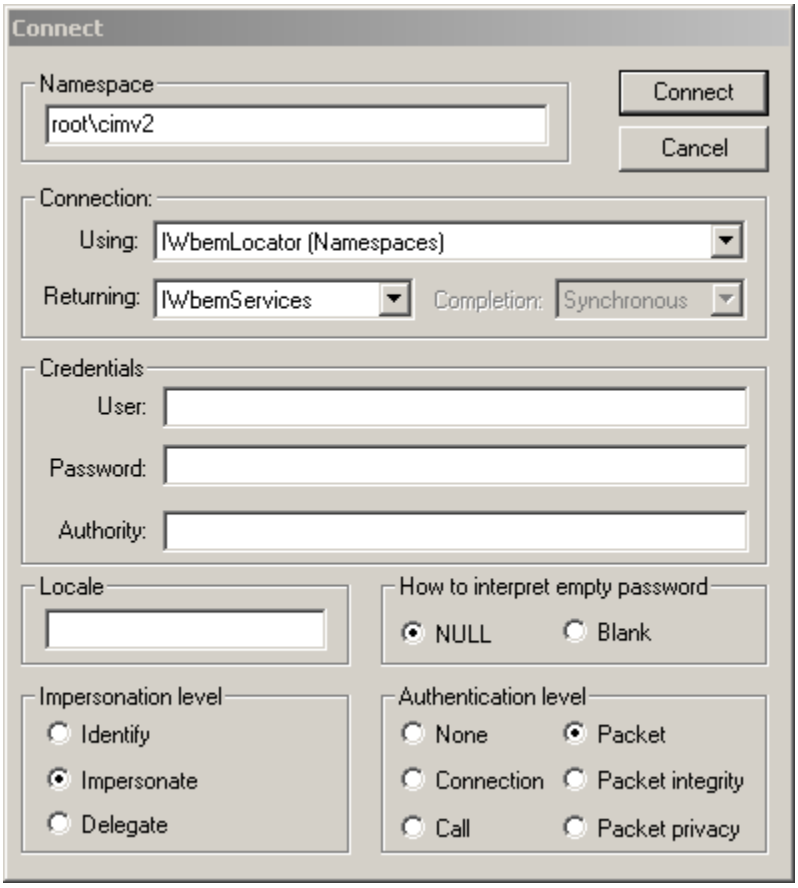

Figure 3.2: Connecting to the WMI namespace.

3. Back on the main dialog box, click Query. Select WQL for the query type, and enter

   ```
   SELECT * FROM Win32_LogicalDisk
   ```

   for the actual query, as Figure 3.3 shows. Click Apply.

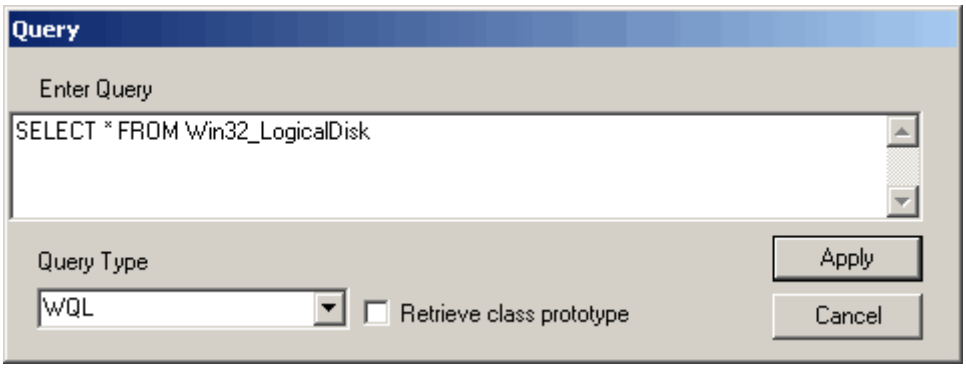

Figure 3.3: Entering a WMI query.

4. You should get something like what Figure 3.4 shows, listing each instance that your query returned.

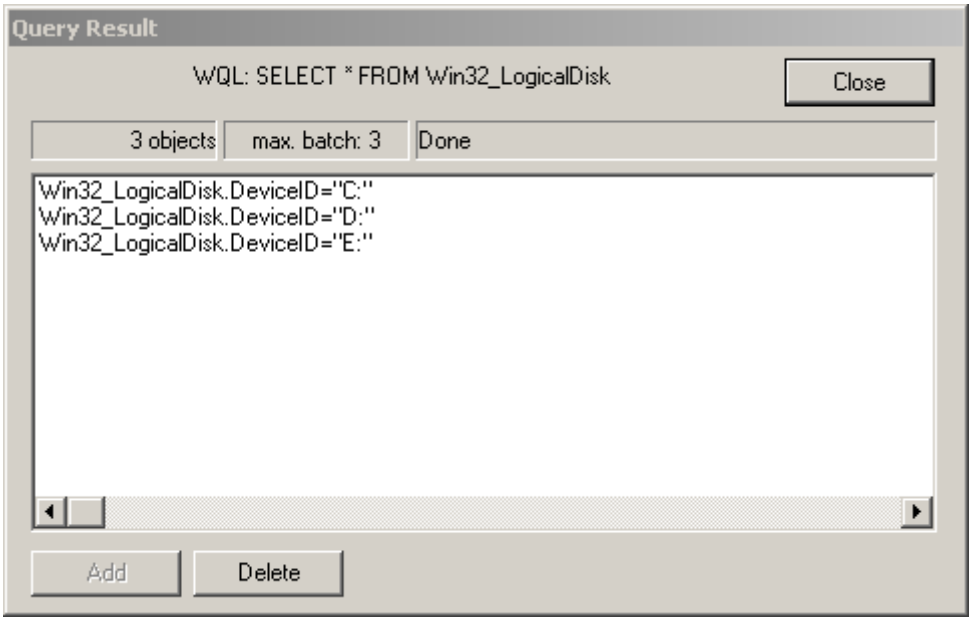

Figure 3.4: Reviewing instances returned by the query.

5. Double-click an instance, and you'll see its properties, as Figure 3.5 shows. Notice the useful information such as FileSystem and FreeSpace.

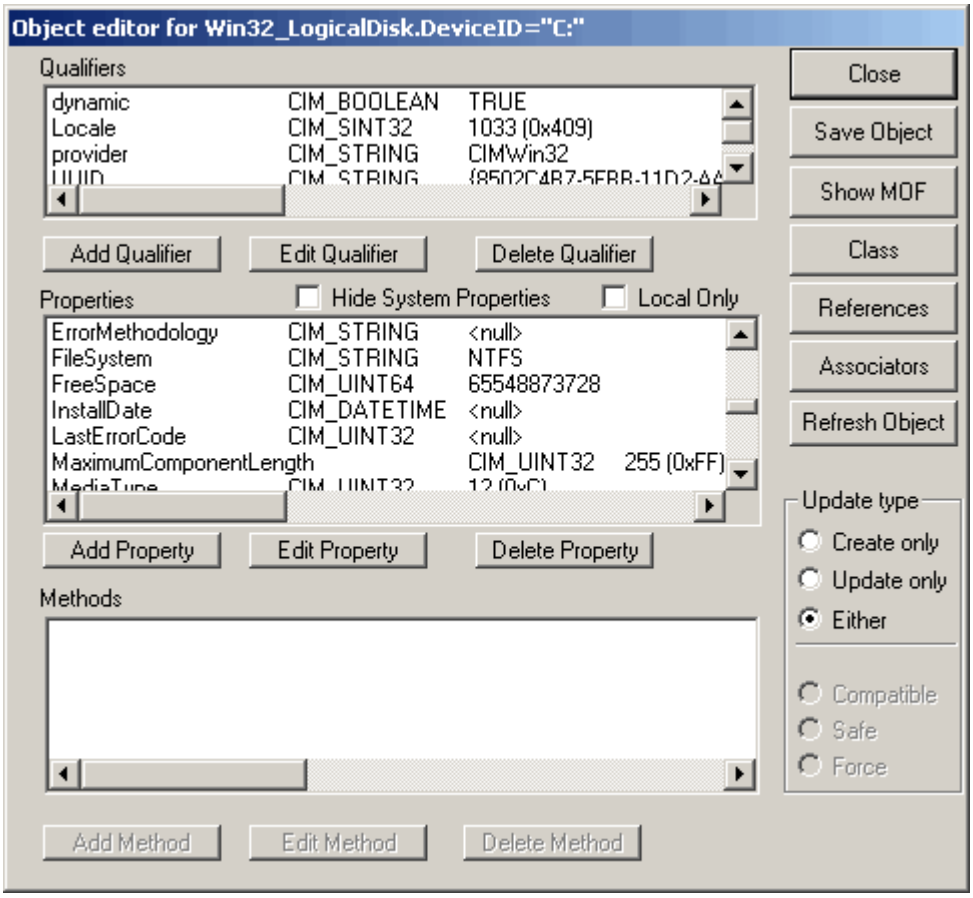

Figure 3.5: Reviewing an instance's properties.

Shazam, you're a WMI guru! Well, almost. This example is meant to illustrate the relationship of classes, instances, and properties, and how they're used to deliver information about your computer.

> ☞ The WBEMTEST tool is built right into Windows XP and WS2K3. The Web-Based Enterprise Management (WBEM) is the standard that WMI implements. Common Information Model (CIM—as in cimv2) is a way of standardizing class and property naming and relationships. The CIM was created by the Desktop Management Task Force (DMTF), an industry task force that includes Microsoft.

Scripting and WMI

Although running queries with WBEMTEST is a great way to test WMI queries, there are more useful WMI administrative applications. As an illustration, consider the Win23_LogicalDisk class script that Listing 3.1 shows.

```
On Error Resume Next
Dim strComputer
Dim objWMIService
Dim propValue
Dim colItems

strComputer = "."
Set objWMIService = GetObject("winmgmts:\\" & _
  strComputer & "\root\cimv2")
Set colItems = objWMIService.ExecQuery("Select * from " & _
  "Win32_LogicalDisk",,48)
For Each objItem in colItems
 WScript.Echo "Access: " & objItem.Access
 WScript.Echo "Availability: " & objItem.Availability
 WScript.Echo "BlockSize: " & objItem.BlockSize
 WScript.Echo "Caption: " & objItem.Caption
 WScript.Echo "Compressed: " & objItem.Compressed
 WScript.Echo "ConfigManagerErrorCode: " & _
  objItem.ConfigManagerErrorCode
 WScript.Echo "ConfigManagerUserConfig: " & _
  objItem.ConfigManagerUserConfig
 WScript.Echo "CreationClassName: " & objItem.CreationClassName
 WScript.Echo "Description: " & objItem.Description
 WScript.Echo "DeviceID: " & objItem.DeviceID
 WScript.Echo "DriveType: " & objItem.DriveType
 WScript.Echo "ErrorCleared: " & objItem.ErrorCleared
 WScript.Echo "ErrorDescription: " & objItem.ErrorDescription
 WScript.Echo "ErrorMethodology: " & objItem.ErrorMethodology
 WScript.Echo "FileSystem: " & objItem.FileSystem
 WScript.Echo "FreeSpace: " & objItem.FreeSpace
 WScript.Echo "InstallDate: " & objItem.InstallDate
 WScript.Echo "LastErrorCode: " & objItem.LastErrorCode
 WScript.Echo "MaximumComponentLength: " & _
  objItem.MaximumComponentLength
 WScript.Echo "MediaType: " & objItem.MediaType
 WScript.Echo "Name: " & objItem.Name
 WScript.Echo "NumberOfBlocks: " & objItem.NumberOfBlocks
 WScript.Echo "PNPDeviceID: " & objItem.PNPDeviceID

 for each propValue in objItem.PowerManagementCapabilities
```

```
  WScript.Echo "PowerManagementCapabilities: " & _
  propValue
 next

 WScript.Echo "PowerManagementSupported: " & _
  objItem.PowerManagementSupported
 WScript.Echo "ProviderName: " & objItem.ProviderName
 WScript.Echo "Purpose: " & objItem.Purpose
 WScript.Echo "QuotasDisabled: " & objItem.QuotasDisabled
 WScript.Echo "QuotasIncomplete: " & objItem.QuotasIncomplete
 WScript.Echo "QuotasRebuilding: " & objItem.QuotasRebuilding
 WScript.Echo "Size: " & objItem.Size
 WScript.Echo "Status: " & objItem.Status
 WScript.Echo "StatusInfo: " & objItem.StatusInfo
 WScript.Echo "SupportsDiskQuotas: " & objItem.SupportsDiskQuotas
 WScript.Echo "SupportsFileBasedCompression: " & _
  objItem.SupportsFileBasedCompression
 WScript.Echo "SystemCreationClassName: " & _
  objItem.SystemCreationClassName
 WScript.Echo "SystemName: " & objItem.SystemName
 WScript.Echo "VolumeDirty: " & objItem.VolumeDirty
 WScript.Echo "VolumeName: " & objItem.VolumeName
 WScript.Echo "VolumeSerialNumber: " & objItem.VolumeSerialNumber
Next
```

Listing 3.1: Example WMI script that involves the Win23_LogicalDisk.

The script that Listing 3.1 shows was produced by the WMI Wizard of the PrimalScript script editor (see http://www.primalscript.com for details). The helpful Scripting Guys at Microsoft provide a free WMI Scriptomatic that is similar to the WMI Wizard in PrimalScript; you can download this Scriptomatic from http://www.microsoft.com/technet/community/scriptcenter/tools/wmimatic.mspx.

The script in Listing 3.1 begins by declaring a few variables. The meat of the script is only a few lines long:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:\\" & _
  strComputer & "\root\cimv2")
Set colItems = objWMIService.ExecQuery("Select * from " & _
  "Win32_LogicalDisk",,48)
```

These lines set a variable named strComputer equal to ".", which happens to be the name of the local computer. You could set this variable to another computer name to pull this information from a remote machine.

Next, the script uses the Set keyword and the GetObject() function to retrieve an object reference and assign it to the variable objWMIService, which is exactly how ADSI works (as you remember from the previous chapter). The object being retrieved is the WMI Service, a background service running on all Win2K and later machines (and on NT machines on which WMI has been installed). The computer name as well as the namespace to connect to—root\cimv2—are passed as part of the GetObject() method.

Finally, the script uses the ExecQuery method to ask the WMI Service to execute a query. The example query should look familiar: it is the same one used in the WBEMTEST example earlier. The variable colItems, then, will contain a collection of instances, just as Figure 3.4 displays a collection of instances. The last part of the script uses a For Each…Next loop to go through each item (instance) in the collection, displaying its properties.

By the way, running this script under CScript.exe, rather than WScript.exe, is more efficient. The reason is that the WScript.Echo statements, under WScript, produce a *lot* of message boxes, which require you to click OK in order to continue. CScript interprets WScript.Echo as simple command-line output, which doesn't require interaction in order to continue.

There's No One, Right Way

There are many ways to connect to WMI. The GetObject() method, used with the winmgmts:// *moniker*, as it's called, is a popular way and you'll see it in a lot of examples. One of the reasons it's popular is that it makes it relatively easy to play funky games with security.

By default, a WMI query such as the previous example script connects to the WMI Service, which runs under the LocalSystem account. Thus, queries get executed as LocalSystem, which might not always be what you want (because, for example, LocalSystem isn't allowed to perform a direct shutdown of a computer).

Suppose you're logged on to ComputerA as a Domain Admin, and you want to shut down ComputerB, which is a domain member. You're an administrator on ComputerB, so you're allowed to do so; what you need WMI to do is *impersonate* your credentials for the shutdown task. The winmgmts:// moniker provides this capability. Listing 3.2 provides an example.

```
strComputer = "computerb"

Set objWMIService = GetObject("winmgmts:" & _
 "{impersonationLevel=impersonate,(Shutdown)}!\\" & _
 strComputer & "\root\cimv2")

Set colOperatingSystems = objWMIService.ExecQuery _
 ("Select * from Win32_OperatingSystem")

For Each objOperatingSystem in colOperatingSystems
 ObjOperatingSystem.Reboot()
Next
```

Listing 3.2: An example script that uses the winmgmts:// moniker.

If you read this script carefully, you'll notice that it appears to be issuing a Reboot() command to multiple instances of Win32_OperatingSystem, as if one computer could actually contain multiple running instances of Windows. Today, computers can't, but someday they might; WMI is built to recognize the existence of multiple active OSs; thus, querying Win32_OperatingSystem could, in theory, return multiple instances.

Notice that some extra text has been tacked into the winmgmts:\\ connection. The complete query is

```
winmgmts:{impersonationLevel=impersonate,(Shutdown)}!\\computerb\
root\cimv2
```

This query tells WMI to impersonate you for the task of shutting down the computer; WMI will try to acquire the necessary permissions on the remote machine and will return an error if it can't.

Alternative Credentials

What the winmgmts://: moniker isn't so good at is providing alternative credentials; meaning ones you're not currently logged on with. In other words, suppose you're logged onto ComputerA as a mere mortal, and want to perform a Domain Admin-style action on ComputerB. The winmgmts:// moniker can't help, but there is another way.

The solution is to fire up a local DLL on the machine running the script. The DLL in question is the WbemScripting.SWbemLocator object, which has the special powers necessary to create a connection to a remote WMI Service and pass alternative credentials. Listing 3.3 provides an example of how to do so.

```
Dim oLocator, oService
Set oLocator = CreateObject("WbemScripting.SWbemLocator")
Set oService = oLocator.ConnectServer( _
 "computerb", "root\cimv2", "Administrator", "Password!")
oService.Security_.impersonationlevel = 3
oService.Security_.Privileges.Add 18
oService.Security_.Privileges.Add 23

Dim oItem
For Each oItem in oService.InstancesOf("Win32_OperatingSystem")
 oItem.Reboot()
Next
```

Listing 3.3: Example script that uses the WbemScripting.SWbemLocator object.

This script does pretty much the same thing that the previous script does, except that this script allows you to pass in alternative credentials. The Locator has a ConnectServer() method, which accepts the server name, namespace, and credentials you want to use. The method returns a reference to the remote WMI Service, which this script references with the variable oService.

The script also messes around with the oddly named Security_ property, which is actually the SWbemSecurity object. One of its properties is impersonationlevel, which is how the WMI Service is told to set an impersonation level. In other words, how should the service impersonate the credentials you've passed along? There are a few values you can provide here:

- 1 means Anonymous, so the credentials you pass in are hidden—that is, not revealed to WMI. WMI probably won't be able to perform whatever you wanted it to with this value.
- 2 means Identify, meaning objects can query the credentials you pass in but not use them. WMI will probably also not work at this level.

- 3 means Impersonate, meaning objects can utilize the credentials you pass in. This value is what WMI usually needs.
- 4 means Delegate, which on Win2K and later allows objects to allow *other* objects to use the credentials you pass in. This value will work with WMI but is usually overkill and may be a security risk.

The script then adds some privileges to the Security_ property's Privileges object. Possible values are:

- 3—Lock physical pages in memory
- 4—Increase the quota assigned to a process
- 5—Create a machine account
- 6—Act as part of the trusted computer base
- 7—Control and view audit messages; required for all tasks reserved for security operators
- 8—Take ownership of an object
- 9—Load or unload a device driver
- 10—Gather profiling information (performance counters) for the system
- 11—Modify system time
- 12—Gather profiling information for a single process
- 13—Increase base priority for a process
- 14—Create a paging file
- 15—Create a permanent object
- 16—Perform backup operations
- 17—Perform restore operations
- 18—Shut down the local system
- 19—Debug a process
- 20—Generate audit log entries
- 23—Shut down a system remotely
- 24—Remove computer from a docking station
- 25—Synchronize directory service data
- 26—Enable computer and user accounts for delegation

This script added both the local and remote shutdown permissions, allowing it to shut down either the local system or a remote system.

Credential Security

Hardcoding usernames and passwords into a script is a horrible, horrible idea. Scripts cannot be reliably protected so that the credentials won't be divulged to someone who shouldn't have them. An alternative is to have the script prompt for credentials using InputBox().

Microsoft provides a free Script Encoder, which is intended to hide the contents of your script from prying eyes while still allowing the script to execute. *This tool is not reliable enough to protect hardcoded credentials.* The encryption keys used by Script Encoder are well-known, and several easily obtainable script decoders exist. Do not imagine for a moment that obscuring credentials with the Script Encoder will protect them; "security through obscurity" is no security at all.

What to Do With WMI

As everything in WMI is based on classes, what you can do with WMI comes down to memorizing every available class—except that there are *hundreds* of classes, and the number grows with each new Microsoft product release. The best place to start to research what you can do with WMI is the MSDN Library at http://msdn.microsoft.com/library. Browse to Setup and System Administration, Windows Management Instrumentation, SDK Documentation, Windows Management Instrumentation, WMI Reference, WMI Classes. This path changes often, so you might need to perform a search.

You'll find sections on Win32 classes, which are useful, and registry classes, which are also useful. Figure 3.6, for example, shows the Win32_OperatingSystem class documentation. As you can see, it lists all of the class' properties as well as its four methods: Reboot, SetDateTime, Shutdown, and Win32Shutdown.

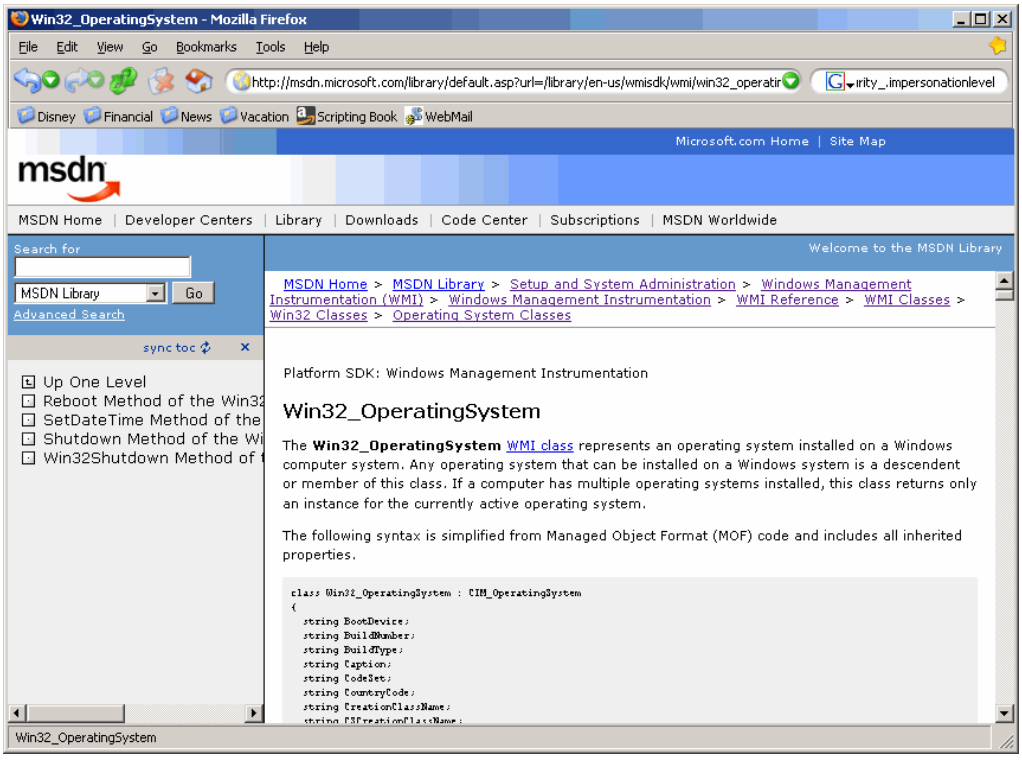

Figure 3.6: Exploring the WMI documentation.

Given the documentation, you can start to see how all WMI scripts start to look alike. For example, consider the script that Listing 3.4 shows, which writes out the current service pack version of each computer in the domain (this script uses the template script used in Chapter 2).

```
‘connect to the root of AD
Dim rootDSE, domainObject
Set rootDSE=GetObject(“LDAP://RootDSE”)
domainContainer = rootDSE.Get(“defaultNamingContext”)
Set oDomain = GetObject(“LDAP://” & domainContainer)

Dim oFSO, oTS
Set oFSO = CreateObject(“Scripting.FileSystemObject”)
Set oTS = oFSO.CreateTextFile(“servicepacks.txt”, True)

‘start with the domain root
WorkWithObject(oDomain)

oTS.Close
MsgBox “Done!”

Sub WorkWithObject(oContainer)
 Dim oADObject
 For Each oADObject in oContainer
  Select Case oADObject.Class
   Case “user”
    ‘oADObject represents a USER object;
    ‘do something with it
    ‘** YOUR CODE HERE**
   Case “computer”
    strComputer = ADSObject.cn
    ‘oADObject represents a COMPUTER object;
    ‘do something with it
    ‘connect to the computer via WMI
    Set oWMIService = GetObject(“winmgmts:” _
     & “{impersonationLevel=impersonate}!\\” & _
     strComputer & “\root\cimv2”)

    ‘retrieve OS class
    Set oWindows = oWMIService.ExecQuery(“SELECT * “ & _
     “FROM Win32_OperatingSystem”)

    ‘output sp level
    For Each oOS in oWindows
     oTS.WriteLine oADObject.Name & “: “ & _
      oOS.Name & “ / “ & oOS.Version & “ SP” & _
      oOS.ServicePackMajorVersion & “.” & _
      oOS.ServicePackMinorVersion
    Next

   Case “organizationalUnit” , “container”
    ‘oADObject is an OU or container...
    ‘go through its objects
    WorkWithObject(oADObject)
  End select
 Next
End Sub
```

Listing 3.4: Example script that writes out the current service pack version of each computer in the domain.

Suppose that you wanted to see how much free space was available on every computer's disk drives. A very similar script emerges; Listing 3.5 shows the new script with the few lines that have been changed from the previous example script in boldface.

```
'connect to the root of AD
Dim rootDSE, domainObject
Set rootDSE=GetObject("LDAP://RootDSE")
domainContainer = rootDSE.Get("defaultNamingContext")
Set oDomain = GetObject("LDAP://" & domainContainer)

Dim oFSO, oTS
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTS = oFSO.CreateTextFile("diskspace.txt", True)

'start with the domain root
WorkWithObject(oDomain)

oTS.Close
MsgBox "Done!"

Sub WorkWithObject(oContainer)
 Dim oADObject
 For Each oADObject in oContainer
  Select Case oADObject.Class
   Case "user"
    'oADObject represents a USER object;
    'do something with it
    '** YOUR CODE HERE**
   Case "computer"
    strComputer = ADSObject.cn
    'oADObject represents a COMPUTER object;
    'do something with it
    'connect to the computer via WMI
    Set oWMIService = GetObject("winmgmts:" _
     & "{impersonationLevel=impersonate}!\\" & _
     strComputer & "\root\cimv2")

    'retrieve OS class
    Set oDisks = oWMIService.ExecQuery("SELECT * " & _
     "FROM Win32_LogicalDisk")

    'output sp level
    For Each oDisk in oDisks
     oTS.WriteLine oADObject.Name & ": " & _
      oDisk.Name & " has " & oDisk.FreeSpace & " bytes free"
    Next

   Case "organizationalUnit" , "container"
    'oADObject is an OU or container...
    'go through its objects
    WorkWithObject(oADObject)
  End select
 Next
End Sub
```

Listing 3.5: Example script to see how much free space was available on every computer's disk drives.

Once you become accustomed to working with WMI, it all starts to look a lot alike. This repetition is great because it means that the time you take to write one WMI-based script will save you a lot of time the next time you need to write one.

WMI Scriptlets

At this point, you're probably itching to see some more WMI in action. WMI can be useful for querying information, changing settings, and performing actions such as restarting a computer. The next few sections will provide scriptlets that demonstrate the possibilities. Keep in mind that these may require some minor modifications—such as different server names—in order to run properly in your environment; don't try to cut and paste them and expect them to work right away in every environment.

Managing Services

Services are something we would all just as soon forget about. They're not readily managed from a central location and they are difficult to manage on a server-by-server basis. VBScript can help, as Listing 3.6 shows.

```
'connect to the computer's WMI provider
sComputer = "server1"
Set oWMIService = GetObject("winmgmts:" _
  & "{impersonationLevel=impersonate}!\\" & _
  sComputer & "\root\cimv2")

'retrieve the MyApp service
Set oService = oWMIService.ExecQuery _
("Select * from Win32_Service WHERE Name" & _
   " = 'myservice'")

   'change the password
   errReturn = oService.Change( , , , , , , _
   , "NeWP@ss#0rD")
```

Listing 3.6: An example script that helps in managing services.

This script connects to WMI on a server named Server1, using WMI impersonation to impersonate the credentials of whoever is running the script. It executes a WMI query that is a bit unlike any you've seen so far. Rather than querying *every* instance of Win32_Service, it queries all instances whose Name property is myservice. You could make that Alerter, Messenger, or some actual service name. The script then executes the service's Change() method, skipping the first seven parameters, which are things we don't want to change right now, and passes in a new password. This new password becomes the password that the service will use to log on. This scriptlet is a great way to update a service to use a new password, making it easier to regularly change service account passwords.

> ☞ If you get an error indicating that "Object doesn't support this property or method: 'oService.Change'," the odds are that oService isn't set to an instance of the Win32_Service class. That can—and will—happen if your query doesn't specify a valid service name.

Of course, this script only works against one computer; it would be more useful if it could run against a list of computers running this particular service. Say, a text file with one computer name per line. Listing 3.7 shows an example script that does so.

```
'open a text file of computer names
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTS = oFSO.OpenTextFile("c:\computers.txt")

Do Until oTS.AtEndOfStream

 'get computer name
 sComputer = oTS.ReadLine

 'connect to the computer's WMI provider
 Set oWMIService = GetObject("winmgmts:" _
   & "{impersonationLevel=impersonate}!\\" & _
   sComputer & "\root\cimv2")

 'retrieve the MyApp service
 Set oService = oWMIService.ExecQuery _
 ("Select * from Win32_Service WHERE Name" & _
    " = 'myservice'")

 'change the password
 errReturn = oService.Change( , , , , , , _
  , "NeWP@ss#0rD")
Loop

oTS.Close
```

Listing 3.7: An example script that manages a list of computers running a particular service.

In Listing 3.7, the modified lines are highlighted in boldface so that you can see what a relatively minor change this is. Want the script to provide feedback about what's going on? You need to make only one simple change, as Listing 3.8 shows.

```
'open a text file of computer names
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTS = oFSO.OpenTextFile("c:\computers.txt")

Do Until oTS.AtEndOfStream

 'get computer name
 sComputer = oTS.ReadLine

 'connect to the computer's WMI provider
 Set oWMIService = GetObject("winmgmts:" _
   & "{impersonationLevel=impersonate}!\\" & _
   sComputer & "\root\cimv2")

 'retrieve the MyApp service
 Set oService = oWMIService.ExecQuery _
 ("Select * from Win32_Service WHERE Name" & _
    " = 'myservice'")

 'change the password
```

```
 errReturn = oService.Change( , , , , , , , _
  , "NeWP@ss#0rD")

 'message
 WScript.Echo "Changed " & sComputer

Loop

oTS.Close
```

Listing 3.8: The same example script with a modification added to enable feedback.

What Happened to Dim?

The first chapter stated that the Dim statement was required to tell VBScript that you were planning on using a particular variable. Well, "required" is a strong term. As you can see from the past few samples, scripts run fine without it. The truth is that Dim is optional—you don't have to tell VBScript up front. If VBScript encounters a variable that it hasn't seen before, it implicitly declares it for you, saving you time. However, Dim is a useful tool that can prevent errors from being overlooked. Read the following script sample carefully:

```
sComputer = InputBox("Computer name?")
sPassword = InputBox("New password for service?")
Set oWMIService = GetObject("winmgmts:" _
   & "{impersonationLevel=impersonate}!\\" & _
   sComputer & "\root\cimv2")
Set oService = oWMIService.ExecQuery _
 ("Select * from Win32_Service WHERE Name" & _
    " = 'myservice'")

 'change the password
 errReturn = oService.Change( , , , , , , , _
  , sPasswpd)
```

Spot the problem? The password you entered is stored in the variable sPassword, but you made a typo in the last line and actually used the contents of variable sPasswpd to set the new password. VBScript hasn't seen sPasswpd before, so it initializes a new, empty variable, and now your service is screwed up. To avoid such mistakes, use Dim:

```
Option Explicit
Dim sComputer, sPassword, oWMIService, oService
sComputer = InputBox("Computer name?")
sPassword = InputBox("New password for service?")
Set oWMIService = GetObject("winmgmts:" _
   & "{impersonationLevel=impersonate}!\\" & _
   sComputer & "\root\cimv2")
Set oService = oWMIService.ExecQuery _
 ("Select * from Win32_Service WHERE Name" & _
    " = 'myservice'")

 'change the password
 errReturn = oService.Change( , , , , , , , _
  , sPasswpd)
```

This modified script will result in an error on the last line of code because you didn't declare sPasswpd. The Option Explicit statement at the beginning of the script tells VBScript to disallow implicit variable creation and to require variables to be announced via the Dim statement. Now you'll spot your typo immediately, thanks to the error, without screwing up any of your service settings.

Now that you have this service-changing example in hand, the next thing you should do is scurry into the WMI documentation to find out what *else* this script might be able to do. I've already mentioned that the Win32_Service class' Change() method has at least seven other arguments: What do they do?

According to the documentation, I can use this method to change:

- The display name
- The path name
- The service type
- The error control type
- The start mode
- Whether desktop interaction is allowed
- The login account
- The login password
- The order in which the service should load
- The dependencies the service has

Whenever you get a WMI example, spend some time exploring what else you can do with the objects, classes, or whatever else you've been introduced to. It's a great way not only to accomplish a current task but also to incrementally learn more and more about VBScript and WMI.

Archive Security Logs

Still waiting for the Microsoft Audit Collection Server (MACS) to show up on your network? Listing 3.9 provides a handy script that will read a text file of computer names, connect to each one, archive its security log to a local file (on the computer running the script, that is), then clear the log to make room for new entries.

```
'Create a FileSystemObject
Set oFS = CreateObject("Scripting.FileSystemObject")

'Open a text file of computer names
'with one computer name per line
Set oTS = oFS.OpenTextFile("c:\computers.txt")

'go through the text file
Do Until oTS.AtEndOfStream

 'get next computer
  sComputer = oTS.ReadLine

 'connect to the WMI provider
 Set oWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate,(Backup,Security)}!\\" & _
 sComputer & "\root\cimv2")

 'query the Security logs
 Set cLogFiles = oWMIService.ExecQuery _
 ("Select * from Win32_NTEventLogFile where " & _
 "LogFileName='Security'")

 'go through the collection of logs
 For Each oLogfile in cLogFiles

  'back up the log to a file
  errBackupLog = oLogFile.BackupEventLog _
  ("c:\logfiles\" & sComputer & "\" & Date() & ".evt")

  'see if an error occurred
  If errBackupLog <> 0 Then

   'one did - display an error
   Wscript.Echo "Couldn't get log from " & sComputer

  Else

   'no error - safe to clear the Log
   oLogFile.ClearEventLog()

  End If
 Next
Loop

'close the input file
oTS.Close
```

Listing 3.9: An example script that reads a text file of computer names, connects to each one, archives its security log, then clears the log to make room for new entries.

Notice the impersonation permission being requested: Backup,Security. This permission is necessary to archive security logs. The script selects every instance of Win32_NTEventLogFile where the name of the log file (the LogFileName property) is Security. It's theoretically possible for that query to return multiple instances of Win32_NTEventLogFile, so you must assume the result of the query will be a collection. Therefore, it makes sense to use a For Each…Next construct to loop through each instance one at a time.

> ☞ If you are receiving access denied errors, your system may be configured to require other special permissions. Generally, the Backup permission suffices for log files, although the Security log in particular often requires the additional Security permission. Your system may be configured to require additional permissions.

Each instance's BackupEventLog() method is called. The only argument this method requires is a filename, so I built one that includes the remote computer's name, along with the current date, for easier identification later. The method returns a value greater than zero if the backup didn't work; specifically, it returns the value 8 if permission is denied, 21 if the archive path is invalid, and 183 if the archive path already exists.

If the value returned is zero, the script clears the log using the ClearEventLog() method; if not, the script outputs an error message indicating that the log couldn't be backed up. Note that in some instances, you may need to pass the path and filename of the event log as an argument of ClearEventLog(); if this is necessary, you can modify the script as follows:

```
'no error - safe to clear the Log
oLogFile.ClearEventLog(oLogFile.Path & oLogFile.FileName)
```

Extended WMI

Another great way to fiddle with WMI and see what it can do is by using the Tweakomatic, another tool from the Microsoft Scripting Guys. You can access this tool at http://www.microsoft.com/technet/community/scriptcenter/tools/twkmatic.mspx. For example, if I wanted to modify a computer's screen saver to require a password, the Tweakomatic tells me that I could use the short script that Listing 3.10 shows.

```
HKEY_CURRENT_USER = &H80000001
strComputer = "."
Set objReg = GetObject("winmgmts:\\" & _
 strComputer & "\root\default:StdRegProv")
strKeyPath = "Control Panel\Desktop"
objReg.CreateKey HKEY_CURRENT_USER, strKeyPath
ValueName = "ScreenSaverIsSecure"
strValue = "1"
objReg.SetStringValue HKEY_CURRENT_USER, _
 strKeyPath, ValueName, strValue
```

Listing 3.10: An example Tweakomatic script to modify a computer's screen saver to require a password.

Notice anything different about the WMI query that Listing 3.10 shows? The difference is in the WMI query, which uses root\default:StdRegProv rather than root\cimv2. What's going on there?

This is where WMI and quantum physics really start to look interchangeable. Microsoft only crams certain WMI classes into the root\cimv2 namespace. Other WMI classes get crammed elsewhere. In this example, it's the Standard Registry Provider that the script is working with, so you must to connect to its namespace. Exchange Server 2000 and Exchange Server 2003 install WMI classes, and they use the root\cimv2\applications\exchange namespace. Or they did. In Exchange Server 2000 SP2 and later, Microsoft started the root\cimv2\applications\exchangev2 namespace, which includes providers named ExchangeDsAccessProvider and ExchangeMessageTrackingProvider.

IIS gets in on the game, too, with the MicrosoftIISv2 namespace (root\microsoftiisv2). SQL Server 2000 has the root\MicrosoftSQLServer namespace. Other Microsoft products add WMI classes, too, and it's cursedly difficult to find them because Microsoft doesn't document them in one convenient place. BizTalk Server uses root\MicrosoftBizTalkServer, for example, and I don't know where you would dig up that information without a search. There is a WMI namespace for SNMP providers, allowing you to interact with the providers via WMI. You can install and uninstall applications using the Windows Installer namespace, and there is even a namespace for performance counters, so you can write scripts that retrieve performance information. Unfortunately, WMI is so broad in scope that it would be impossible to address them all in this guide.

You can use the WBEMTEST tool, or even the WMI Scriptomatic, to browse the available classes (to a point; they don't query every available namespace). But those tools will only list classes available on the *local machine*; thus, if you want to see things like Exchange classes, you have to run the tool on a machine that contains those classes—namely, an Exchange Server system.

If you're eager to see what namespaces are available on a particular computer, run the following script (I recommend running it under CScript because it outputs a lot of information):

```
sComputer = "."
Call EnumerateNamespaces("root")
Sub EnumerateNamespaces(sNamespace)
 WScript.Echo sNamespace
 Set oWMIService = GetObject("winmgmts:\\" & _
  sComputer & "\" & sNamespace)
 Set cNamespaces = oWMIService.InstancesOf("__NAMESPACE")
 For Each oNamespace In cNamespaces
  Call EnumerateNamespaces(sNamespace & _
   "\" & oNamespace.Name)
 Next
End Sub
```

My WS2K3 machine returns the following interesting entries, giving you some idea of the scope WMI encompasses:

- Root\SECURITY—for working with security
- Root\perfmon—for working with performance counters
- Root\RSOP—for working with Resultant Set of Policy (RSOP) in Group Policy
- Root\snmp—for working with SNMP
- Root\MSCluster—for working with Microsoft Cluster Server
- Root\cimv2—for working with core Win32 classes
- Root\Applications\MicrosoftIE—for working with Internet Explorer
- Root\MicrosoftActiveDirectory—for working with Active Directory
- Root\MicrosoftIISv2—for working with IIS 6.0
- Root\Policy—for working with policies
- Root\MicrosoftDNS—for working with DNS
- Root\MicrosoftNLB—for working with Network Load Balancing
- Root\registry—for working with the registry

If you repeat this chapter's very first exercise, using WBEMTEST, and substitute these namespace for root\cimv2, you'll be able to check out the classes in these namespaces. Instead of clicking Query, click Enum Classes. Leave the Superclass name blank, click Recursive, and click OK. You should get a list of all available classes in the namespace.

Summary

WMI, ADSI, and VBScript work together to provide a well-rounded, robust set of administrative tools. Between them, there is not much you can't do with regard to remote management. However, there are some advanced techniques that can make scripting a bit more powerful, and I'll take a look at some of them in the next chapter.

Chapter 4: Advanced Scripting

Aside from the amazing things you can do with WMI and ADSI, scripting can provide a lot of additional functionality for making administration easier. For example, you can work with databases in a script, which gives you the ability to log WMI information into a SQL Server or Access database. In addition, the ability to run scripts on remote machines lets you extend your administrative reach and scope across your entire enterprise. In this chapter, I'll touch on these and other advanced topics, giving you a head start toward making your scripts more powerful and flexible.

Remote Scripting

We've already explored a form of remote scripting—running a script that affects remote computers from your computer. WMI and ADSI, in particular, are useful for this type of remote scripting. As Figure 4.1 illustrates, the script executes on one computer but performs operations against one or more remote computers. Typically, the script executes under the authority of the user account running the script. However, some technologies—including WMI—provide the means to specify alternative credentials, which the script can use when connecting to remote machines.

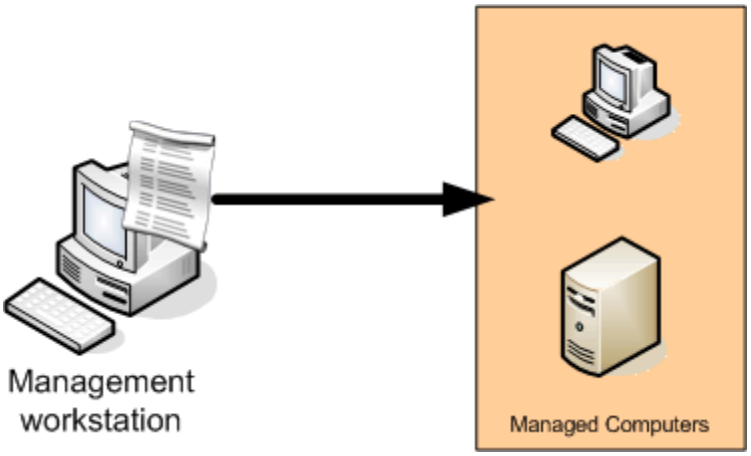

Figure 4.1: Basic remote scripting.

Another type of remote scripting is made possible by using the WshController object. As Figure 4.2 shows, this object actually copies a script from your machine to the remote machines, which execute the script independently. WshController allows you to monitor the remote script execution for errors and completion.

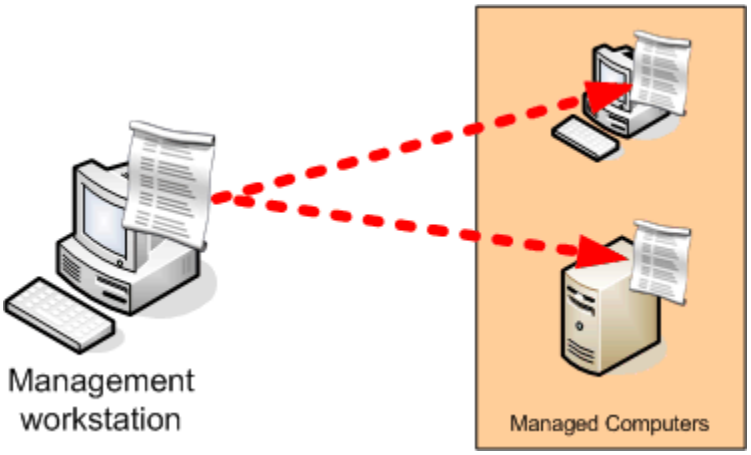

Figure 4.2: Using the WshController object for remote scripting.

The WshController Object

WshController is created just like any other object: by using the CreateObject() function. WshController has just one method, CreateScript(). This method returns a WshRemote object, which allows you to interact with a remote script. Suppose you have a script named C:\Script.vbs on your local machine that you want to run on a computer named ClientB. You would use a script similar to the following example:

```
Dim oController, oRemote
Set oController = WScript.CreateObject("WSHController")
Set oRemote = oController.CreateScript("c:\Script.vbs", _
 "ClientB")
oRemote.Execute
```

> Remote scripting is available only in the latest version of the Windows Script Host (WSH), version 5.6, and on NT 4.0 Service Pack 3 (SP3) and later versions of Windows. In general, you must be a local administrator on the remote machine, and remote WSH must be enabled in the registry of the remote machine. You can do so by navigating the registry to HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows Script Host\Settings, and adding a key named Remote as a REG_SZ (String) value. Set the value to 1 to enable remote WSH and 0 to disable it. It's disabled by default for security reasons. WSH 5.6 must be installed on both the machine sending the script and the machine that is to run it (the remote machine).

Not too difficult. Of course, with just that code, you won't be able to track the remote script's status. Add the following to provide tracking:

```
Do While oRemote.Status = 0
 WScript.Sleep 100
Loop
MsgBox "Remote execution completed with status " & oRemote.Status
```

The Status property can be either 0 or 1: 0 means the remote script is still running, and 1 means it has completed. The WshRemote object has two methods: Execute and Terminate. We've explore how the Execute method is used. The Terminate method can be used to end a still-running script, and this method requires no parameters.

WshRemote also provides a child object, WshRemoteError. This child object provides access to errors occurring in remote scripts. Using it is a bit more complicated and requires the use of the default WScript object's ConnectObject() method.

WScript.ConnectObject

You've already seen how the intrinsic WScript object's CreateObject() and GetObject() methods are used. The ConnectObject method is similar to GetObject() in that it deals with a remote object. Rather than retrieving a reference to a remote object, however, ConnectObject allows you to synchronize object events. As we've previously explored, objects have three basic members:

- Properties, which describe the object and modify its behavior
- Methods, which make the object perform some action
- Collections, which provide access to child objects

There is actually one other type of member: an event. *Events* provide an object with the means to inform your script when something occurs. For example, buttons in the Windows user interface (UI) fire an event when someone clicks them. This event being fired tells the underlying code that the button was clicked, allowing the underlying code to take whatever action it's supposed to take when the button is clicked. The following example demonstrates this concept:

```
Dim oController, oRemoteScript
Set oController = WScript.CreateObject("WSHController")
Set oRemoteScript = oController.CreateScript("me.vbs","Server1")

WScript.ConnectObject oRemoteScript, "remote_"
oRemoteScript.Execute

Do While oRemoteScript.Status <> 2
 WScript.Sleep 100
Loop

WScript.DisconnectObject oRemoteScript
```

This example script closely follows the previous example to create a remote script, execute it, and wait until it finishes. But this example adds the ConnectObject method to synchronize the remote script's events with this script's events. Any remote events will be fired back to this script. This script needs to contain a subroutine, or *sub*, prefixed with "remote_", because that is what the script told ConnectObject to look for when events occur. You could add the following:

```
Sub remote_Error
 Dim oError
 Set oError = oRemoteScript.Error
 WScript.Echo "Error #" & oError.Number
 WScript.Echo "At line " & oError.Line
 WScript.Echo oError.Description
 WScript.Quit
End Sub
```

The DisconnectObject method is used when the script is over to cancel the connection between the remote script and the script shown here.

Remote Scripting Limitations

Remote scripting does have some limitations. Remote scripts shouldn't use InputBox(), MsgBox(), or WScript.Echo to produce output because remote scripts aren't given an interactive desktop to work with. Any output from a remote script will need to be written to a text file on a file server or some other central location where you can retrieve it and look it over.

Remote scripts also have some security limitations. Generally speaking, they'll run under the context of the LocalSystem account, although that does vary between Windows versions and may change in service packs for Windows XP and later versions to a less-powerful account. Also, because scripts run under that context, they may have difficultly accessing anything in the local profile of a user. For example, accessing registry keys in HKEY_CURRENT_USER won't necessarily connect to the currently logged on user (because the script isn't running under that user's context), which can create unexpected results for your scripts. If you absolutely need a script to run as the logged on user, assign the script as a logon script.

Database Scripting

Scripting is completely compatible with Microsoft's universal data access technology, called ActiveX Data Objects (ADO). As the name implies, ADO utilizes objects, so your scripts will use CreateObject() to instantiate these objects and assign them to variables.

Making Data Connections

ADO uses a Connection object to provide a connection to data sources, such as Access databases, SQL Server databases, Excel spreadsheets, and more. Creating the Connection object is simple:

```
Set oConn = CreateObject("ADODB.Connection")
```

You have a couple of options, however, for specifying the data to which you want to connect. The easiest—although, as I'll explain, not quite the most efficient—is to use an Open Database Connectivity (ODBC) Data Source Name (DSN). Windows XP and later computers provide an ODBC Control Panel in the Administrative Tools group; other versions of Windows provide this option from the Control Panel instead. Figure 4.3 shows the ODBC application and a list of configured DSNs for the current user (*System* DSNs are available for all users of the computer).

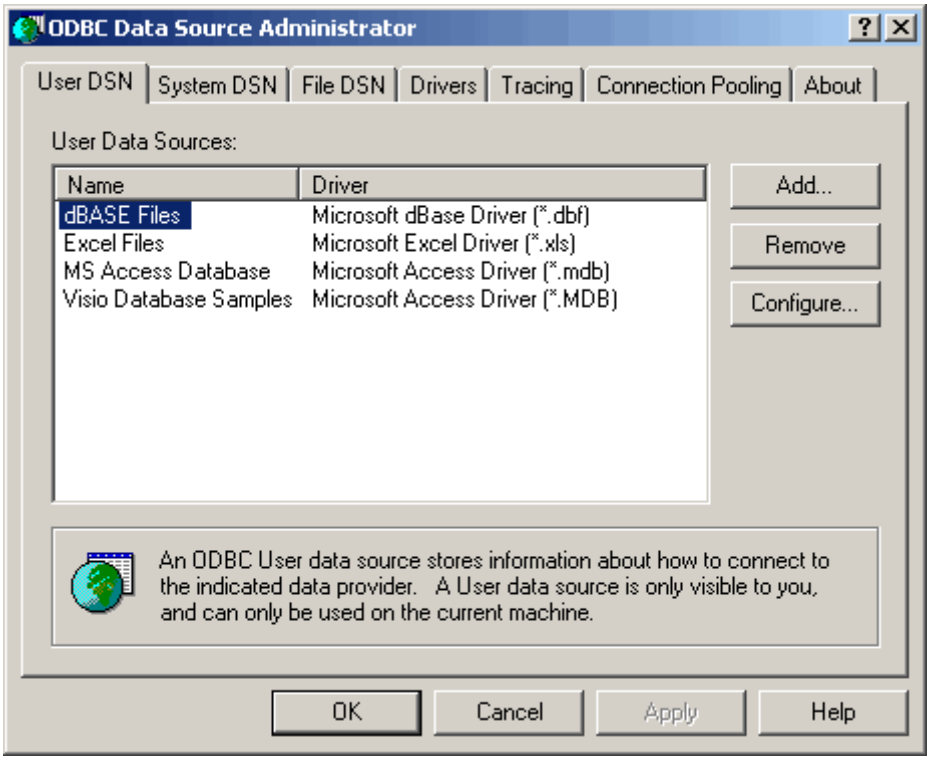

Figure 4.3: The ODBC panel.

For example, to add a DSN for an Access database, click Add, and select the Access driver from the list. Specify a name for your data source (such as "MyData"), and click Select to specify an Access MDB database. Once the DSN is created, you add code to open that DSN in your script:

```
oConn.Open "MyData"
```

There are a couple of downsides to using DSNs:

- They must be individually configured on each computer on which the script will run
- They access data through the older ODBC software in Windows—although this access method won't create a performance problem for most scripts, it isn't the most efficient way; you're effectively asking ADO to access data by using a technology that ADO itself (actually, the underlying OLE DB technology) was supposed to supersede

An alternative method is to use a native call and tell ADO which driver to use, what data to open, and so forth, all using a *connection string*. Sample drivers (or *providers,* in ADO terminology) include:

- Microsoft.Jet.OLEDB.4.0 (Access)
- SQLOLEDB (SQL Server)
- ADSDSOObject (Active Directory—AD)

Connection strings look a bit different depending on the driver used, because each driver requires slightly different information, for example:

- For an Access database: "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=*folder\file.mdb*"
- For a SQL Server database: "Provider=SQLOLEDB; Data Source=*server*; Initial Catalog=*database*; User ID=*user*; Password=*password*"
- For AD: "Provider=ADSDSOObject; User ID=*user*; Password=*password*"

Thus, using this method, you would open the connection as follows for an Access database:

```
oConn.Open "Provider=Microsoft.Jet.OLEDB.4.0; " & _
 "Data Source=c:\databases\mydatabase.mdb"
```

This part of the process is the most difficult part of working with ADO. Once the connection is open, you can simply start querying and modifying data.

A Quick Database Lesson

Data in a database—whether it's an Excel spreadsheet or a SQL Server database—is organized into several logical components. A *table* is the main logical unit. Databases can consist of multiple tables. In an Excel file, each worksheet is treated as an individual table.

A *row* or *entity* or *record* contains a single entry in a table (for example, rows in an Excel spreadsheet or rows in an Access database table). A *column* or *domain* or *field* represents a single piece of information. For example, a column in an Excel spreadsheet might contain user names, and another column contains domain names for those users. ADO works with rows and columns to provide you with access to the data in a database.

Querying and Displaying Data

You'll probably find it easier to query data by using SQL-style queries, even when you're not accessing a SQL Server database; ADO understands the SQL query language and makes it pretty easy to use SQL with any type of data source. The basic syntax for a query looks like this:

```
SELECT column, column, column
FROM table
WHERE comparison
```

For example, suppose you have an Excel spreadsheet like the one that Figure 4.4 shows.

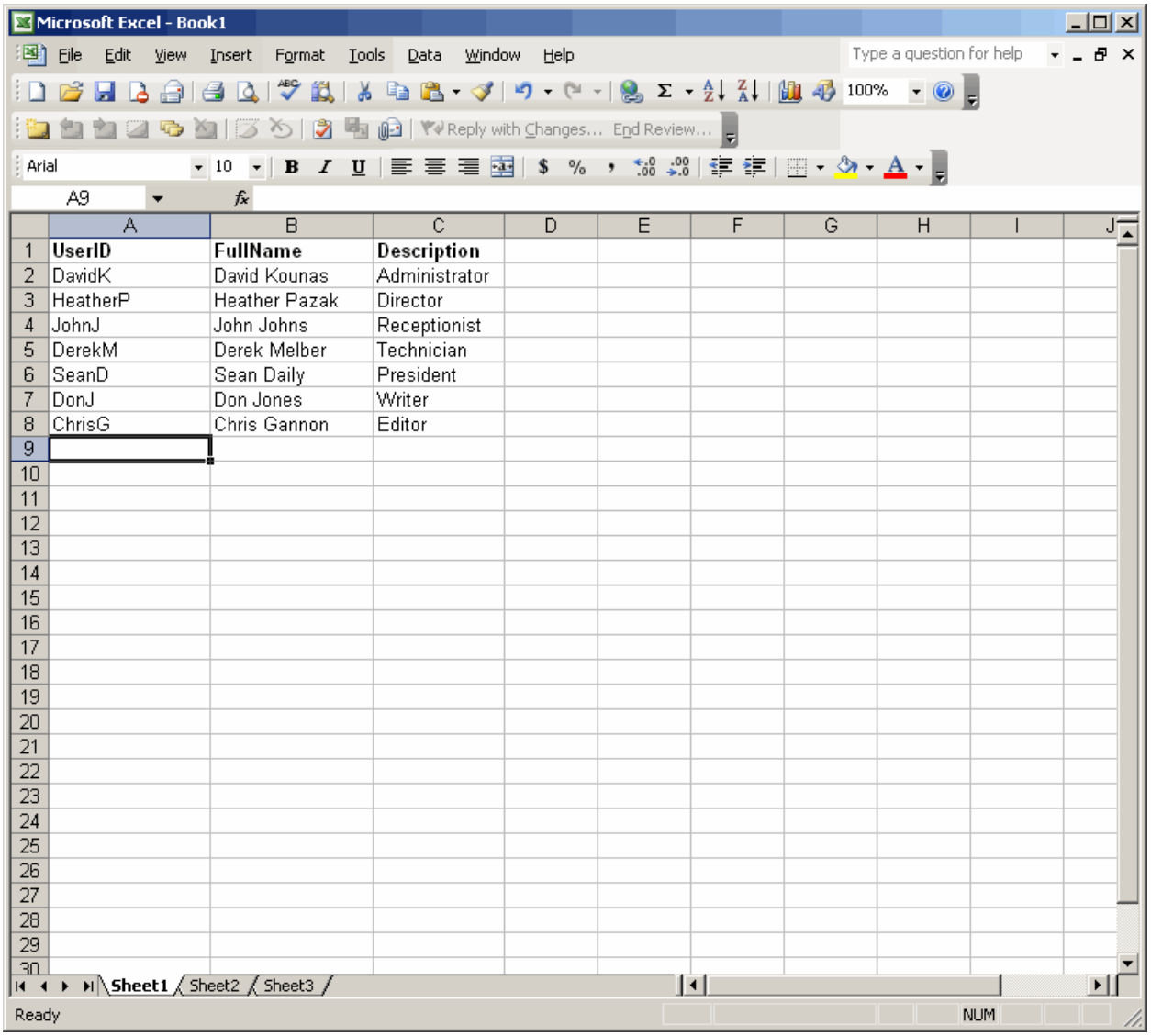

Figure 4.4: Example Excel spreadsheet.

The table name in this case is Sheet1$ (Excel adds a dollar sign to the end of the worksheet name). There are three columns: UserID, FullName, and Description.

> ☞ In general, it's easiest to have table and column names that don't contain spaces or punctuation. However, if they do, you need to surround them in square brackets: [User ID], for example.

Suppose you wanted to query the UserID for every row in the table:

```
SELECT UserID FROM [Sheet1$]
```

If you only wanted the UserID of users whose description was "Writer," you would use this:

```
SELECT UserID FROM [Sheet1$] WHERE Description = 'Writer'
```

The results of your query are a set of rows—or as database people like to say, a set of records. In ADO parlance, that is a *recordset*, and it's represented by a Recordset object. To implement your query in VBScript, assuming a Connection object named oConn had been created and opened:

```
Set oRS = oConn.Execute("SELECT UserID FROM [Sheet1$]")
```

That leaves you with a Recordset object containing the specified rows and columns; in this case, it would contain seven rows and one column (assuming we're querying the Excel spreadsheet I showed you).

You should first determine whether your recordset contains anything. Recordset objects contain an internal pointer, which points to the current record. The objects also provide methods for moving the pointer to the next record, and properties for telling you when you've moved the pointer to the beginning of the file (recordset) or the end of the file (recordset). Those two properties, BOF (for the beginning of the file) and EOF (for the end of the file), will both be True for an empty recordset. So you can use the following comparison:

```
If oRS.EOF and oRS.BOF Then
 'no records
Else
 'records
End If
```

You can access the data in the recordset by simply referring to the column names. For example, the following will display the User ID for the current record:

```
WScript.Echo oRS("UserID")
```

Finally, you can move to the next record with this:

```
oRS.MoveNext
```

> ✎ In case you're wondering, there is a MovePrevious method. However, the default type of recordset returned by the Connection object's Execute() method is an efficient *forward-only* recordset, meaning that once you've used MoveNext to advance to the next record, you can't move back.
>
> Covering the vast complexity and flexibility of the entire set of ADO objects is a bit beyond the scope of this guide. However, you can check out the ADO documentation in the MSDN Library at http://www.microsoft.com/msdn; just look for the Data Access category.

Thus, writing a script that outputs all of the user IDs by using a DSN named "Excel," might look like this:

```
Dim oConn, oRS
Set oConn = CreateObject("ADODB.Connection")
oConn.Open "Excel"
Set oRS = oConn.Execute("SELECT UserID FROM [Sheet1$]")
If oRS.EOF and oRS.BOF Then
 WScript.Echo "No records returned"
Else
 Do Until oRS.EOF
  WScript.Echo "UserID: " & oRS("UserID")
  oRS.MoveNext
 Loop
End If

oRS.Close
oConn.Close
```

Notice that I threw in two lines of code to close the recordset and connection when the script ends. You don't strictly need to do so because VBScript will more or less do it automatically when the script ends, but it's a good practice.

As an extended example, the script that Listing 4.1 shows queries a DSN named "Excel" (which is assumed to be an Excel spreadsheet) and creates new users in an NT or AD domain. This script assumes that the Excel spreadsheet contains columns named UserID, FullName, and Description, much like the example I showed you earlier. The script creates passwords for the new users, and writes those passwords to a text file for distribution.

```
' PART 1: Open up the Excel spreadsheet
' using ActiveX Data Objects
Dim oCN
Set oCN = CreateObject("ADODB.Connection")
oCN.Open "Excel"

Dim oRS
Set oRS = oCN.Execute("SELECT * FROM [Sheet1$]")


' PART 2: Get a reference to the
' Windows NT domain using ADSI
Dim oDomain
Set oDomain = GetObject("WinNT://DOMAIN")


' PART 3: Open an output text file
' to store users' initial passwords
Dim oFSO, oTS
Set oFSO = CreateObject("Scripting.FileSystemObject")
```

```
Set oTS = oFSO.CreateTextFile("c:\passwords.txt",True)


' PART 4: For each record in the recordset,
' add the user, set the correct user
' properties, and add the user to the
' appropriate groups

' create the necessary variables
Dim sUserID, sFullName, sDescription
Dim sPassword, oUserAcct

' now go through the recordset one
' row at a time
Do Until oRS.EOF

  ' get the user information from this row
  sUserID = oRS("UserID")
  sFullName = oRS("FullName")
  sDescription = oRS("Description")

  ' make up a new password
  sPassword = Left(sUserID,2) & DatePart("n",Time) & _
   DatePart("y",Date) & DatePart("s",Time)

  ' create the user account
  Set oUserAcct = oDomain.Create("user",sUserID)

  ' set account properties
  oUserAcct.SetPassword sPassword
  oUserAcct.FullName = sFullName
  oUserAcct.Description = sDescription

  ' save the account
  oUserAcct.SetInfo

  ' write password to file
  oTS.Write sUserID & "," & sPassword & vbCrLf


  ' PART 5: All done!
  ' release the user account
  Set oUserAcct = Nothing

  ' move to the next row in the recordset
  oRS.MoveNext

Loop


' PART 6: Final clean up, close down
oRS.Close
oTS.Close
WScript.Echo "Passwords have been written to c:\passwords.txt."
```

Listing 4.1: An example script that queries a DSN named "Excel."

Note that you'll need to insert the correct domain name, which I've boldfaced, in order for the script to work. In an AD domain, users will be created in the default Users container.

Modifying Data

ADO isn't limited to pulling data from a database; it can modify and add information, too. There are a couple of ways to do so. The most straightforward, perhaps, is to issue a data modification query, using the Connection object's Execute method. This method is the same method that returns a Recordset object when used with a SELECT query; with a data modification query, it doesn't return anything. Here's how it works:

```
'Delete rows
oConn.Execute "DELETE FROM table WHERE criteria"

'Change rows
oConn.Execute "UPDATE table SET column=value WHERE criteria"

'Add rows
oConn.Execute "INSERT INTO table (column, column) " & _
 "VALUES ('value', 'value')"
```

For example, let's take the previous example, which creates user accounts and writes their passwords to a file. If the Excel spreadsheet had an additional column named Password, we could modify the script to save the passwords right into the spreadsheet instead of into a separate file. Listing 4.2 shows the lines of code that get removed in ~~strikethrough~~, and the changed lines in boldface.

```
' PART 1: Open up the Excel spreadsheet
' using ActiveX Data Objects
Dim oCN
Set oCN = CreateObject("ADODB.Connection")
oCN.Open "Excel"

Dim oRS
Set oRS = oCN.Execute("SELECT * FROM [Sheet1$]")


' PART 2: Get a reference to the
' Windows NT domain using ADSI
Dim oDomain
Set oDomain = GetObject("WinNT://DOMAIN")


~~' PART 3: Open an output text file~~
~~' to store users' initial passwords~~
~~Dim oFSO, oTS~~
~~Set oFSO = CreateObject("Scripting.FileSystemObject")~~
~~Set oTS = oFSO.CreateTextFile("c:\passwords.txt",True)~~


' PART 4: For each record in the recordset,
```

```
‘ add the user, set the correct user
‘ properties, and add the user to the
‘ appropriate groups

‘ create the necessary variables
Dim sUserID, sFullName, sDescription
Dim sPassword, oUserAcct

‘ now go through the recordset one
‘ row at a time
Do Until oRS.EOF

  ‘ get the user information from this row
  sUserID = oRS(“UserID”)
  sFullName = oRS(“FullName”)
  sDescription = oRS(“Description”)

  ‘ make up a new password
  sPassword = Left(sUserID,2) & DatePart(“n”,Time) & _
   DatePart(“y”,Date) & DatePart(“s”,Time)

  ‘ create the user account
  Set oUserAcct = oDomain.Create(“user”,sUserID)

  ‘ set account properties
  oUserAcct.SetPassword sPassword
  oUserAcct.FullName = sFullName
  oUserAcct.Description = sDescription

  ‘ save the account
  oUserAcct.SetInfo

  ‘ write password to database
  oCN.Execute “UPDATE [Sheet1$] SET Password = ‘“ & _
   sPassword & “‘ WHERE UserID = ‘“ & sUserID & “‘“


  ‘ PART 5: All done!
  ‘ release the user account
  Set oUserAcct = Nothing

  ‘ move to the next row in the recordset
  oRS.MoveNext

Loop


‘ PART 6: Final clean up, close down
oRS.Close
oTS.Close
WScript.Echo “Passwords have been written to the database.”
```

Listing 4.2: Example script that writes passwords to an existing Excel spreadsheet rather than a separate file.

There is another way to change data when you’re using a Recordset object: Simply change the columns similar to the way you read data from them. When you’re finished, use the Recordset’s Update method. Listing 4.3 shows the entire script again, modified to use this new method.

```
‘ PART 1: Open up the Excel spreadsheet
‘ using ActiveX Data Objects
Dim oCN
Set oCN = CreateObject(“ADODB.Connection”)
oCN.Open “Excel”

Dim oRS
Set oRS = oCN.Execute(“SELECT * FROM [Sheet1$]”)


‘ PART 2: Get a reference to the
‘ Windows NT domain using ADSI
Dim oDomain
Set oDomain = GetObject(“WinNT://DOMAIN”)


‘ PART 3: Open an output text file
‘ to store users’ initial passwords
Dim oFSO, oTS
Set oFSO = CreateObject(“Scripting.FileSystemObject”)
Set oTS = oFSO.CreateTextFile(“c:\passwords.txt”,True)


‘ PART 4: For each record in the recordset,
‘ add the user, set the correct user
‘ properties, and add the user to the
‘ appropriate groups

‘ create the necessary variables
Dim sUserID, sFullName, sDescription
Dim sPassword, oUserAcct

‘ now go through the recordset one
‘ row at a time
Do Until oRS.EOF

  ‘ get the user information from this row
  sUserID = oRS(“UserID”)
  sFullName = oRS(“FullName”)
  sDescription = oRS(“Description”)

  ‘ make up a new password
  sPassword = Left(sUserID,2) & DatePart(“n”,Time) & _
   DatePart(“y”,Date) & DatePart(“s”,Time)

  ‘ create the user account
  Set oUserAcct = oDomain.Create(“user”,sUserID)

  ‘ set account properties
  oUserAcct.SetPassword sPassword
  oUserAcct.FullName = sFullName
  oUserAcct.Description = sDescription

  ‘ save the account
  oUserAcct.SetInfo

  ‘ write password to database
  oRS(“Password”) = sPassword
```

```
  oRS.Update

  ‘ PART 5: All done!
  ‘ release the user account
  Set oUserAcct = Nothing

  ‘ move to the next row in the recordset
  oRS.MoveNext

Loop


‘ PART 6: Final clean up, close down
oRS.Close
oTS.Close
WScript.Echo “Passwords have been written to the database.”
```

Listing 4.3: Example script that uses the Recordset object.

The Recordset object also supports an AddNew method for adding new rows, and a Delete method for deleting rows; check out the ADO documentation for details on using these methods in your scripts.

Windows Script Files

So far, all the scripts I’ve used in this guide are designed to be put into VBS files and run more or less as-is. WSH provides another file type, WSF (for Windows Script File), which is a more powerful and flexible format. WSF files are XML-formatted and provide better capabilities for creating scripts that accept command-line parameters. Here’s a brief example:

```
<package>
    <job id=“VBS”>
    <?job debug=“true”?>
        <script language=“VBScript”>
           WScript.Echo “This is VBScript”
        </script>
    </job>
    <job id=“JS”>
    <?job debug=“true”?>
        <script language=“JScript”>
           WScript.Echo(“This is JScript”);
        </script>
    </job>
</package>
```

Notice that the script contains several distinct elements:

- The entire script is contained in a <package>.
- Multiple <job> elements can exist, each with its own ID.
- Each <job> can contain a <script>, which can be in VBScript or JScript (or any other installed scripting language).
- The script itself is contained between the <script> tag and the </script> tag.

This example doesn't provide much beyond what a plain text file could do. The real fun of the WSF format comes with additional sections. Consider the example that Listing 4.4 shows.

```
<job>
    <runtime>
        <named
            name="server"
            helpstring="The server to run the script on"
            type="string"
            required="true"
        />

    <description>
    This script connects to a remote server and restarts it
    </description>

    <example>
    Example: Restart.wsf /server:servername
    </example>

    </runtime>
<script language="VBScript">
  'insert script here
</script>
</job>
```

Listing 4.4: Example WSF script.

This script defines a new <runtime> section, which contains several helpful sub-elements. The first is <named>. This element defines a named command-line argument. In this case, the name of the argument is "server," and it is intended to be a string value. It is required; if the script is executed without this argument, WSH won't allow the script to run. If this script were named "restart.wsf," you would execute it by running

```
restart.wsf /server:servername
```

from a command-line. The type can be "string," "Boolean," or "simple." In the case of "simple," the argument doesn't take a value.

Within your script, you can use the following code to display an automatically-generated "help file" for your script based on its arguments' "helptext" parameters and the <example> and <description> elements:

```
WScript.Arguments.ShowUsage
```


The <example> and <description> elements are just text and should be self-explanatory.

Users can also display the help text by running the script with the standard /? argument.

Within your script, you would access these arguments by using the WshArguments object. Using the above WSF file as an example, you might do something like the following in the main body of the script:

```
oArgs = WScript.Arguments.Named
sServerName = oArgs.Item("server")
```

The variable sServerName would now contain the value specified for the "server" argument. Because WSF files provide this great functionality for defining arguments, and because they can automatically produce a "help file" screen, it's a great format for using VBScript to create your own command-line utilities.

Signing Scripts

I recommend enabling WSH 5.6's script signature verification policies on all computers in your environment (read more about this feature on the Windows Scripting home page at http://www.microsoft.com/scripting and at http://www.ScriptingAnswers.com). This feature, when used properly, will prevent all unsigned scripts from executing, helping to prevent script-based viruses.

In general, you enable the feature by editing the registry. Navigate to HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows Script Host\Settings, and add a REG_DWORD value named "TrustPolicy." Set the value to 2 to fully enable signature verification.


There are also HKEY_LOCAL_MACHINE-related registry keys that affect verification policy, and you may need to modify these in Windows XP and WS2K3 machines in order to fully enable the verification policy. I provide an ADM template in the Downloads section of http://www.ScriptingAnswers.com; use this template to centrally configure and manage the signature verification policy settings via Group Policy objects (GPOs). Simply import the ADM template into a GPO and configure it as desired in the User and Computer configuration sections of the GPO.

However, in order to make sure *your* scripts run, you will need to sign them using a digital certificate issued for the purposes of code signing. You can purchase such a certificate from commercial certification authorities (CAs) such as VeriSign (http://www.verisign.com) and Equifax (http://www.equifax.com), or issue one from an internal CA, if your organization has one.


Your client and server computers must be configured to trust the publisher of the certificate you use. Consult the Windows documentation for information about importing a new trusted certificate publisher, if necessary; most commercial CAs are trusted by default.

Microsoft provides an object called Scripting.Signer that can take a certificate and sign a script. Note that signing a script marks the script with the signature, meaning you can't change the script without invalidating the signature (and preventing the script from running). If you need to modify the script, you will need to re-sign it.

The following code sample shows how to write a script that signs other scripts:

```
Set oSigner = CreateObject("Scripting.Signer")
sFile = InputBox("Path and filename of script to sign?")
sCert = InputBOx("Name of certificate to use?")
sStore = InputBox("Name of certificate store?")

oSigner.SignFile(sFile, sCert, sStore)
```

You'll simply need to know the name of your certificate and the certificate store in which the certificate is installed.

Summary

In this chapter, I've introduced you to some advanced VBScript topics, including script signing and security, flexible WSF files, remote scripting and script events, and ADO. Combined with what you've learned about VBScript's basics, WMI, and ADSI, you should be able to start producing some useful scripts on your own.

I'll leave you with some links to additional online resources that are designed specifically for Windows administrative scripting:

- My Web site at http://www.ScriptingAnswers.com
- The Microsoft TechNet Script Center at http://www.microsoft.com/technet/community/scriptcenter/default.mspx
- Clarence Washington's excellent Win32 Script Repository at http://cwashington.netreach.net/
- The Desktop Engineer's Junk Drawer at http://desktopengineer.com/
- Windows Scripting on MSN Groups at http://groups.msn.com/windowsscript/_homepage.msnw?pgmarket=en-us
- *Windows & .NET Magazine's* Windows Scripting Solutions at http://www.winnetmag.com/WindowsScripting/
- Chris Brooke's *Scripting for MCSEs* column at http://www.mcpmag.com/columns/columnist.asp?ColumnistsID=7

I think you'll find that the interest in administrative scripting is growing and that there is a constantly expanding set of resources for you to take advantage of. Good luck, and enjoy!