- Table of Contents
- Examples

Managing Windows® with VBScript and WMI

By Don Jones

Publisher: Addison Wesley
Pub Date: March 26, 2004
ISBN: 0-321-21334-3
Pages: 640
Slots: 1.0

"Finally, a step-by-step VBScripting book to make you look like a programmer without the time and sweat! Don't waste your time searching the Internet for examples-this book does it for you!"
-Greg A. Marino, Senior Systems Engineer/Consultant, Westtown Consulting Group, Inc.

Visual Basic Scripting (VBScript) and Windows Management Instrumentation (WMI) are vital tools for systems administrators grappling with the increasing complexity of Windows technologies. However, busy admins have been without a straightforward guide to scripting...until now.

Managing Windows(R) with VBScript and WMI explains how Windows administrators can effectively use VBScript to automate common administrative tasks and simplify complex ones. Detailed coverage of security concerns provides admins with the means for safely using VBScript in Windows environments. The book is organized around the problems you face daily, with reusable examples and coverage of Windows NT, Windows 2000, Windows XP, and Windows 2003.

This user-friendly reference demystifies scripting and then shows you how to produce new scripts from scratch. You will be producing useful scripts right away as you study the VBScript language and learn how to control nearly every aspect of the Windows operating system with WMI and the Active Directory Services Interface (ASDI). You will be able to build your own administrative Web pages and use advanced scripting technologies such as script encryption, scripting components, and script security. The book closes with still more ready-made example scripts accompanied by complete line-by-line explanations. The CD includes all the code from the book and trial versions of PrimalScript 3.0 and VbsEdit. A companion Web site provides updates and errata.

Inside you will find answers to such questions as:

- How do you write effective logon scripts? Chapter 11
- How do you write scripts that query and modify user and group information? Chapter 16
- How can you query the IP addresses from multiple network adapters in multiple remote computers? Chapter 19

- How can you design, write, run, test, and debug your own administrative Web pages? Chapter 24
- How can you reuse code between various scripts? Chapter 25

< Day Day Up >

- Table of Contents
- Examples

Managing Windows® with VBScript and WMI

By Don Jones

Publisher: Addison Wesley
Pub Date: March 26, 2004
ISBN: 0-321-21334-3
Pages: 640
Slots: 1.0

Copyright

Library of Congress Cataloging-in-Publication Data

Jones, Don. Managing Windows with VBScript and WMI / Don Jones. p. cm. Includes index. ISBN 0-321-21334-3 (pbk. : alk. paper) 1. Microsoft Windows (Computer file) 2. Operating systems (Computers) 3. VBScript (Computer program language) I. Title. QA76.76.O63J6623 2004 005.4'4682—dc22 2003026069

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10—CRS—0807060504

First printing, March 2004

Dedication

This is for Chris.

For seven years, you've given me nothing but love, support, and encouragement, and I can never thank you enough.

Praise for Managing Windows® with VBScript and WMI

"This is the 'must-have' scripting guide for administrators of all levels. It shows what you need to get the job done and how."

—Joseph Niemi, Systems Engineer

"Finally, a step-by-step VBScripting book to make you look like a programmer, without the time and sweat! Don't waste your time searching the Internet for examples; this book does it for you!"

—Greg A. Marino, Senior Systems Engineer and Consultant
Westtown Consulting Group, Inc.

"Each chapter is like having a personal tutor at your side through the iterative process of scripting, no matter what your experience level. Whether you are writing complex scripts or simply automating repetitive tasks, I found this book truly hits the mark as the right tool to get the job done."

—Joel Yoker, Program Manager
Microsoft Corporation

"This book will provide any technical reader with a real-world understanding of how to use VBScript with WMI. The examples are clear and pertinent to the needs of the day-to-day system administrator."

—Bob Reselman, Principal
Cognitive Arts and Technologies

Preface

Microsoft introduced Visual Basic, Scripting Edition—commonly known as VBScript—in the mid-'90s, positioning it as a native replacement for Windows' aging command-line batch language, which was based on Microsoft's earliest operating system, MS-DOS. VBScript was intended to be easy to learn, powerful, and flexible. The language was included as an add-on to Windows 95 and Windows NT 4.0, was an optional installation component included in Windows 98, and was included in all editions of Windows Me, Windows 2000, Windows XP, and Windows Server 2003.

Software developers immediately seized upon VBScript for Web programming, particularly in Active Server Pages, Microsoft's rapid-development programming framework for the Web. However, Windows administrators—one of VBScript's initial target audiences—were left cold. VBScript seemed to be much more complicated than administrators' beloved MS-DOS-based batch language, and many didn't see the need to learn an entirely new batch language.

When Windows 2000 and Active Directory came along, however, administrators found that Windows administration had become a great deal more complex. Suddenly, administrators were searching for Resource Kit and other utilities that offered automated administration, especially for repetitive tasks. Active Directory enabled the use of VBScript for logon and logoff scripts, which seemed to promise more advanced use environment manipulation. At around the same time, Microsoft's naïveté in releasing a powerful language like VBScript with absolutely no security controls resulted in a huge wave of high-impact VBScript-based viruses, forcing administrators to lock down their environments and remove VBScript as an option both for viruses and for administrative tools.

As a regular speaker at some of the country's top technical conferences that focus on Windows technologies, including MCP TechMentor, the past few years I've given half- and full-day sessions on VBScripting for Windows administrators, and the sessions have been incredibly popular. In these sessions, I try to provide just enough VBScript experience to make scripting possible, and then concentrate on accomplishing common administrative tasks with VBScript. I also cover the security concerns of VBScript and provide administrators with the means for safely using VBScript in their environments. This book is essentially a written form of those sessions, greatly expanded with more coverage of Windows Management Instrumentation and other advanced topics, and with more coverage of VBScript security issues and resolutions.

I'm not out to turn you into a programmer. In fact, one of the real successes of VBScript is that you don't need to be a programmer to use it. Most of what you'll be doing in this book involves using VBScript to tell Windows to do things for you; you'll be able to ignore much of VBScript's complexity, using it as a sort of electronic glue to combine various operating system functions.

Who Should Read This Book?

The only assumption I have about you is that you already know how to administer some version of Microsoft Windows. You'll find that most of the material in this book is suitable for Windows NT, Windows 2000, and Windows Server 2003 environments, and it will continue to be useful through future versions of Windows. I do not assume that you have any background in programming, and I'm not going to give you a programming background.

You should have a desire to learn how to use what I call "the batch language of the twenty-first century" and a wish to move away from clumsier—and often more complex—batch files based on the MS-DOS batch language. Although some folks like to refer to batch files as scripts, I don't; and when you see how easy and flexible VBScript is, you'll understand why!

How to Use This Book

You can read this book in order from the Introduction to the Appendix. However, if you already have some experience with VBScript, or if you just want to dive right into the more complete example scripts, you can skip around as much as you like. I've organized this book in the same way that I organize my live VBScripting sessions at conferences, so you may feel that it's some time before you really get into the meat of scripting. I assure you, though, that each example in this book—starting in Chapter 1—is focused on Windows administration. You'll get your feet wet right away!

I've also included In This Chapter elements at the start of each chapter and Coming Up elements at the end of each chapter. These are brief paragraphs that are intended to help set the stage and help you decide if you need to read a particular chapter or not. They'll also help you decide which chapter to read next based on your individual needs and interests. I hope that these elements—along with the cross-references I've included in each chapter—will help you zip straight to the scripting information that you need most.

To help you decide where to start, here's a brief overview of each chapter.

Part I: Introduction to Windows Administrative Scripting

Part I serves as an introduction to the world of scripting and provides you with a methodology for approaching administrative tasks from a scripting standpoint. One of the most difficult parts about producing new scripts from scratch is the "Where do I start?" factor, and I'll provide you with a framework for figuring that out every time.

Chapter 1: Scripting Concepts and Terminology

As I've already implied, administrative scripting isn't hard-core programming. Instead, it's using VBScript as a sort of electronic glue to secure various bits of the Windows operating system together. In this chapter, I'll introduce you to those various bits and set the stage with some basic terminology that you'll use throughout this book.

Chapter 2: Running Scripts

Writing a script isn't much fun if you can't run the script, and so this chapter will focus on the technologies used to execute scripts. You might be surprised to learn how many different Microsoft products support scripting. In this chapter, I'll show you how far your scripting skills can really take you. I'll also introduce you to some scripting tools that can make writing and debugging scripts a bit easier.

Chapter 3: The Components of a Script

In this chapter, I'll present a complete administrative script, and then break it down line-by-line to explain its various components. Although this chapter isn't necessary to learning administrative scripting, it will help you write scripts that are more reliable and easier to troubleshoot.

Chapter 4: Designing a Script

As I've mentioned already, one of the toughest aspects about scripting can be figuring out where to start. In this chapter, I'll provide you with a framework that you can use as a starting point for every new scripting project. I'll also introduce you to some concepts that many scripting books ignore, such as planning for errors and creating a useful "resource kit" of script components that you can reuse throughout your scripting projects.

Part II: VBScript Tutorial

Here's your official crash course to the VBScript language: just enough to make administration via script a possibility! The best part is that I won't use the trite "Hello, world" examples that books for software developers often start out with. Instead, I'll make every example useful to you as a Windows administrator. That means you'll be producing simple, useful scripts at the same time you're learning VBScript. What could be better?

Preparing to Use This Book

Before you dive in, you should make sure that your computers are ready for VBScript. Fortunately, any computer with Windows 2000 or later is ready to go out of the box, and I'll assume that you're doing your development work on either a Windows 2000–, Windows XP–, or Windows Server 2003–based computer.

Typographical Elements

Books on programming can benefit a great deal from easy-to-understand typestyles and elements like the ones I'll explain here. These typestyles and elements are designed to make the text easier to follow and to call your attention to special concerns.

Boldfaced type will be used to set off material that should be typed into the computer; boldfaced also will be used to signify onscreen menu and/or dialog box selections. For example, "select Run from the Start menu, type wbemtest, and click OK" sets off the menu selection, what you need to type onscreen, and what should be clicked from the dialog box.

Blocks of code and code lines that appear within the text will appear in a monospaced font, as in, "to change the contents of a variable, you can use something like Var1 = Trim(Var1)."

TIP

Tips will provide shortcuts and other "insider advice" about scripting that you'll find valuable.

NOTE

Notes will provide cautions and other clarifications that will help you avoid problems or further clarify complex concepts.

I'll also direct you to material that more thoroughly explains particular concepts, VBScript commands, and so forth. Although I'm not a big fan of flipping back and forth through a book, these cross-references will allow you to remain focused within each chapter and will guide you to more detailed explanations, when appropriate.

Finally, there are times when it is necessary to present an extended explanation of something that isn't critical to the task at hand. In those cases, I'll use a sidebar. A sidebar is a cue that the information is useful, but it's not really key to the main text; you're welcome to skip the sidebar and come back to it later if you like.

Sidebars

Sidebars make it easier to cover slightly off-topic information without distracting you from the main text.

➤➤ Sample Scripts

Obviously, a book on scripting is going to have many code listings. To make these as useful as possible, I introduce each example script with a special head style, and present it in a listing by itself with no comments.

Listing P.1. A Sample Script**. Accepts a name, and then redisplays that name.**

```
'Get the user's name

sName = InputBox("What is your name?")



'Display the user's name

MsgBox "Your name is " & sName
```

Acknowledgments

Books never seem to write themselves, no matter how much money I leave out for the elves at night. Fortunately, behind every author like myself stands an incredible team of professionals, and the folks behind this book were truly at the top of their industry. So, special thanks to my literary agent at Studio B, Neil Salkind, and my acquisitions editor at Addison-Wesley, Sondra Scott: Thanks so much for your hard work in taking this book through the acquisitions process! I'd also like to offer a special thanks to technical editors Bob Reselman and Scott Worley for what was probably the most professional, thorough review of any book I've written. And a big thanks to all the other behind-the-scenes folks at Pearson Education who worked so hard to bring this book to market.

On a more personal level, I'd like to thank the folks in my life who make a project like this worthwhile: My parents, Rhonda and John; my ferrets, Clyde, Ziggy, Buffy, Pepper, and little Tigger; and my close and supportive friends, Sonya, Chris, Spencer, George, and Tom. I'd also like to thank my business partner, Derek Melber, who has been so patient while I've been devoting time to this book. I'd also like to thank Keith, Dian, and Kris at Microsoft® Certified Professional Magazine for all of their personal and editorial support. Finally, I'd like to thank all the folks at Microsoft that I've enjoyed working with through the years: Amy, Dr. Todd, Mary Beth, Ben, Harold, Kris, Christine, Kim, Steve, Deb, Chuck, Peggy, Bucky, Barb, Alice, Joel, and Andy.

Don Jones
BrainCore.Net, LLC
February 2004

About the Author

Don Jones is an independent consultant, speaker, and author, and a founding partner of BrainCore.Net, the world leader in exam development and exam delivery technologies for the information technology industry. Don's focus has always been on high-end Windows administration topics, and his recent books include Special Edition Using Microsoft® .NET Enterprise Servers (published by Que) and Windows® Server 2003 Weekend Crash Course (published by Wiley). Don is also the creator and series editor of the Delta Guides, a series of books published by Sams designed to help experienced administrators quickly come up to speed on new versions of Microsoft server technologies. Don coauthored the first book in the series, Windows Server 2003 Delta Guide. Don is now firmly based in Las Vegas, Nevada, after spending more than two years traveling the United States in a 40-foot RV.

Part I: Introduction to Windows Administrative Scripting

Chapter 1. Scripting Concepts and Terminology

IN THIS CHAPTER

Completely new to scripting? This is the chapter for you! You'll learn how scripts work, what they are, and how Windows uses them. You'll also learn about key security issues, which will be covered in more detail in later chapters.

In the past few years, scripting has become increasingly popular with Windows administrators. Visual Basic, Scripting Edition—commonly known as VBScript—has become especially popular, due to its ease of use and incredible flexibility. Unfortunately, most books on scripting seemed to be focused toward developers, or at least toward Windows administrators with a strong software development background. The result is that most administrators think that scripting is too complex for them, which simply isn't true. In this book, I'll introduce you to scripting from a purely administrative standpoint, starting with this chapter, where I'll explain exactly what I mean by "scripting," and how it all fits into Windows administration.

What Is Scripting?

Scripting means different things to different people. Some folks, for example, would define script as any series of computer commands that are executed in a sequence, including so-called scripts written in the MS-DOS batch language. These batch files were the mainstay of administrative automation for many years, and many administrators still rely heavily upon them today. Other people define scripting as small computer programs written in a high-level scripting language, such as VBScript.

Nobody's really wrong, and scripting can mean all of these things. Personally, I fall into the latter camp, believing that it has to be written in VBScript, JavaScript, or some other high-end language to earn the name scripting. Although batch files are certainly a means of automating administrative tasks in Windows, they don't really have the power or flexibility of modern scripting languages—nor should you expect them to. Batch files are based on a command language that's two decades old!

For the purposes of this book, scripting will refer to the act of creating, executing, and utilizing small computer programs that are written in a high-end scripting language, specifically VBScript.

Script Hosts

Scripts start out life as simple text files. Try this: Open Windows Notepad on a Windows XP computer, and type the following text:

```
Set SNSet = GetObject("winmgmts:").InstancesOf ("Win32_OperatingSystem")

for each SN in SNSet

 MMsgBox "The serial number for the installed OS is: " & SN.SerialNumber

next
```

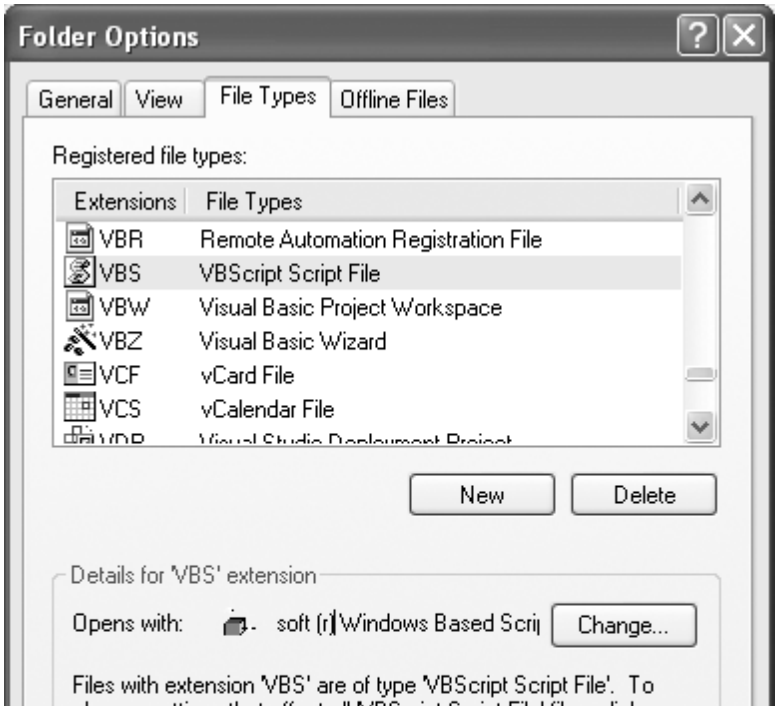
Save the file as "SampleScript.vbs." Be sure to include the filename in double quotation marks, or Notepad will append a TXT filename extension. Now, locate the file in Windows Explorer. Make sure it has a VBS filename extension and double-click it. Provided you're running Windows XP and VBScript hasn't been disabled on your computer, you should see a small dialog box containing the serial number of your operating system. Congratulations, you've just scripted!

NOTE

For the time being, you don't need to worry about how this script does what it does. In later chapters, I'll explain what each of these four lines of code accomplishes. If you just can't wait, jump to Chapters 17 through 19, where I demonstrate how to use Windows Management Instrumentation to retrieve serial numbers and other operating system information.

What actually happens when you double-click the VBS file? You can find out easily enough. From any Windows Explorer window, select Folder Options from the Tools menu. Select the File Types tab and locate VBS in the list. As shown in Figure 1.1, the VBS filename extension is associated with the Microsoft Windows Based Script Host. Whenever you double-click a VBS file, Windows fires up the Script Host, tells it which script you double-clicked, and lets the Script Host run the script. It's similar to what happens when you double-click a DOC file: Windows fires up Microsoft Word, tells it which file to open, and your document appears.

Figure 1.1. File association for the VBS file type



ActiveX Scripting Languages

VBScript is just one of many ActiveX Scripting Languages. These languages are written to a specification developed by Microsoft, and scripts written in these languages can be executed by WSH. Each ActiveX Scripting Language is implemented by a scripting engine. Usually, this DLL file interfaces with WScript.exe to interpret scripts, one line at a time, so that WSH can execute them. Microsoft maintains two ActiveX Scripting Languages: VBScript and JScript. JScript is Microsoft's implementation of ECMAScript, which is the industry-standard version of Netscape's JavaScript scripting language.

NOTE

Ignoring company copyrights, trade names, and other legal matters, JScript, ECMAScript, and JavaScript are more or less interchangeable terms.

The scripting engines are maintained separately from WSH and carry their own version numbers. However, both the latest version of VBScript and JScript are included with the basic WSH installation, so you don't need to worry about getting them individually.

Other companies have produced ActiveX Scripting Languages, too. For example, VideoScript is an independent scripting language that works with WSH (www.videoscript.com). PerlScript and LiveScript are other popular ActiveX Scripting Languages.

Scripting languages all have a few common characteristics.

- They are interpreted. That means the scripting engine reads each line of script, one at a time, and then executes it. Execution requires the WSH to translate the scripted instructions into native Windows API calls. Interpreted languages are slower than compiled languages like Visual Basic 6.0, where the compiler translates the entire program into native Windows code all at once, saving time later when the program is executed.
- They are text based. In other words, you can create scripts with a simple text editor like Notepad. The downside is that anyone can read your script with Notepad, too. Most software applications' code is compiled into a native binary format, making it very difficult for end-users to read the code. Microsoft does offer an encryption utility (discussed in Chapter 27) that allows you to protect your source code from prying eyes.
- They are native. In other words, your scripts will only execute on Windows, because WSH itself will only execute on Windows. Contrast this with languages like Java, which can be compiled and executed on any platform for which a Java Virtual Machine (JVM) is available.
- They are easy to deploy. Unlike compiled Visual Basic 6.0 applications, scripts don't usually require a bunch of DLLs and other files that you have to deploy, register, and so forth. Scripts can generally be copied from one computer to another and executed as-is.

Perhaps the most powerful feature of VBScript is its capability to interface with Microsoft's Component Object Model.

VBScript and .NET: What Does the Future Hold?

The Component Object Model (COM)

Software developers have always been encouraged to develop reusable code. Imagine that you created some piece of code that retrieves the TCP/IP settings of a remote computer. Many administrators might want to use that code again. So how do you make your code available to them in an easy-to-use way?

Microsoft's answer is COM, the Component Object Model. COM is a specification that describes how code can be packaged into objects, making them self-contained, easy (relatively speaking) to deploy, and easy for other developers to use. Physically, COM objects are usually implemented in DLL files—which, if you check out the contents of a Windows computer's System32 folder, should tell you how pervasive COM is!

VBScript is completely capable of utilizing COM objects. That's a powerful feature, because most of Windows' functionality—and most other Microsoft applications' functionality—is rolled up into COM components. Working with e-mail, Active Directory, Windows Management Instrumentation, networking, the registry, and more is all possible through COM components, and therefore through VBScript. I'll cover objects in more detail, including examples of how to use them in script beginning in Chapter 5, and show you how to really take advantage of them in Chapter 11.

VBScript is even capable of creating COM components. That means you can use VBScript to create your IP-retrieval software, package that software as a COM component, and distribute it to other administrators. This feature of scripting is called Windows Script Components. Chapter 25 is all about modular script programming, including Windows Script Components.

Critical Scripting Security Issues

Sadly, Microsoft implemented VBScript without much thought for the consequences. Windows XP, Microsoft's newest client operating system, shipped with full scripting capability built-in and enabled by default. The power of VBScript can be used not only for beneficial administrative tasks, but also for malicious hacking, and many viruses are based on VBScript or another ActiveX Scripting Language.

Administrators have reacted to the security threat of scripts in a number of ways.

- Deleting WScript.exe. Unfortunately, this doesn't work on Windows 2000 or later, because WScript.exe is under Windows File Protection. Delete it and it just comes back.
- Disassociating the VB, VBS, JS, and other WSH file extensions, or re-associating them to simply open in Notepad rather than in WSH. This effectively disables scripting.
- Deploying antivirus software, such as Norton AntiVirus, which detects script execution and halts it.

Regrettably, disabling scripting usually disables it for good, meaning you can't use scripting for logon scripts, administrative tasks, and other beneficial purposes. There's a middle road that you can take however, which authorizes only certain scripts for execution. This middle road helps protect you against scripts written by hackers, while still allowing scripts to be used for administrative and logon purposes.

Fortunately, Microsoft's come to the table with security improvements that can make scripting safe again, and I've devoted Chapter 28 to the topic of scripting security.

Review

VBScript is one of many available ActiveX Scripting Languages. The scripts that you write are executed by the Windows Script Host (WSH), which is physically implemented as WScript.exe and available for (or included with) all 32-bit Windows operating systems. VBScript—like other ActiveX Scripting Languages—is especially powerful because it can interface with COM, Microsoft's Component Object Model. COM allows VBScript to be infinitely extended to perform other functions, including the majority of the Windows operating system's functions. In fact, COM integration sets VBScript apart from other so-called scripting technologies like old MS-DOS-style batch files.

However, VBScript does bring up some important security issues that you'll need to learn to deal with in your environment. Microsoft's regrettable lack of planning when it comes to scripting has resulted in a huge number of script-based viruses, making scripting a tool for both good and evil. Nonetheless, you can learn to configure your environment so that only approved ("good") scripts run, allowing you to use the power and flexibility of script-based administration, while protecting your environment from malicious scripts.

COMING UP

Chapter 2 focuses on running scripts, editing scripts, and writing scripts. You'll learn about basic and more advanced script authoring tools, and the various ways that you can launch scripts within Windows.

Chapter 2. Running Scripts

IN THIS CHAPTER

It is time to start running—and more importantly, editing—scripts. In this chapter, you'll learn the various ways to execute a script, and learn about the ins and outs of creating and editing scripts. You'll also have your first experience with an administrative script!

Suppose you have several scripts ready to run—what do you do with them? Do you load them into Visual Basic and compile them? How do you distribute them to your users for use as logon scripts? What about when you're ready to start writing your own scripts? What tools are available, and how well do they work? This chapter is designed to introduce you to your scripting toolbox, the bits you'll need to write, run, edit, and debug your administrative scripts.

Windows Script Host

The most common way to run scripts is to use WScript.exe, the graphical version of the Windows Script Host (WSH), which I introduced in Chapter 1. WScript is registered to handle common scripting file extensions, so simply double-clicking a .VB or .VBS file will normally execute WScript.exe, then ask it to execute the double-clicked script.

To see WScript in action, try this:

1. Right-click your desktop and select New; then point to Text File.
2. Rename the new text file to Sample1.vbs.
3. Right-click the file and choose Edit. By default, Windows registered Notepad as the handler for the Edit action, so a blank Notepad window will open.
4. Type WScript.Echo "Displaying Output" and save the file.
5. Close Notepad.
6. Create another new text file on the desktop, and name this one Sample2.vbs.
7. Edit Sample2.vbs and enter the following:

```
Wscript.Echo "Here we go..."


Dim V

V = InputBox("What is your name?")

MsgBox "Hello, " & V
```

These aren't terribly complex scripts, but they'll serve to illustrate some important concepts. First, double-click Sample1.vbs. If your system is properly configured, you'll see a dialog box like the one in Figure 2.1. Click OK on the dialog box to dismiss it and end the first script. Now, double-click Sample2.vbs. It'll start with a similar dialog box, as shown in Figure 2.2. Then, as shown in Figure 2.3, you'll be prompted to enter your name.

Figure 2.1. Basic graphical dialog box from a script

Command-Line Scripts

Most of the time, you'll likely use WScript.exe to execute your scripts, and when I refer to WSH I'll generally do so as a nickname for WScript. However, WSH consists of one other executable, Cscript.exe, which is used to execute scripts on a command line.

The difference with Cscript is that it doesn't provide any non-graphical means of collecting user input. In other words, although you can use a Cscript script to display command-line output, you can't use it to get input from the command-line window. Try this and you'll see what I mean:

1. Open a command-line window.
2. Change to your Desktop folder.
3. Enter Cscript sample1.vbs.

You should see something like Figure 2.5: a basic command-line prompt, with "Displaying Output" shown in the command line. That's the work of WScript.Echo: When executed by WScript.exe, Echo creates a dialog box. When executed by Cscript.exe, Echo outputs to the command line. This allows you to create scripts that can be run graphically or from a command line. Scripts written with this technique will appear to be natively written for each environment.

Figure 2.5. Executing Sample1.vbs with Cscript.exe

Now try the same thing with Sample2.vbs. At first, you'll notice a command line like the one in Figure 2.6, simply displaying the output of WScript.Echo as in the previous example. However, when Cscript hits the InputBox function, it switches into graphical mode, as shown in Figure 2.7, just like WScript did. Finally, the MsgBox command also forces Cscript to display a dialog box, as shown in Figure 2.8 and exactly as WScript did—only WScript.Echo is dual mode, working differently in WScript or Cscript. Everything else defaults to a graphical mode of operation.

Figure 2.6. Command-line output of WScript.echo

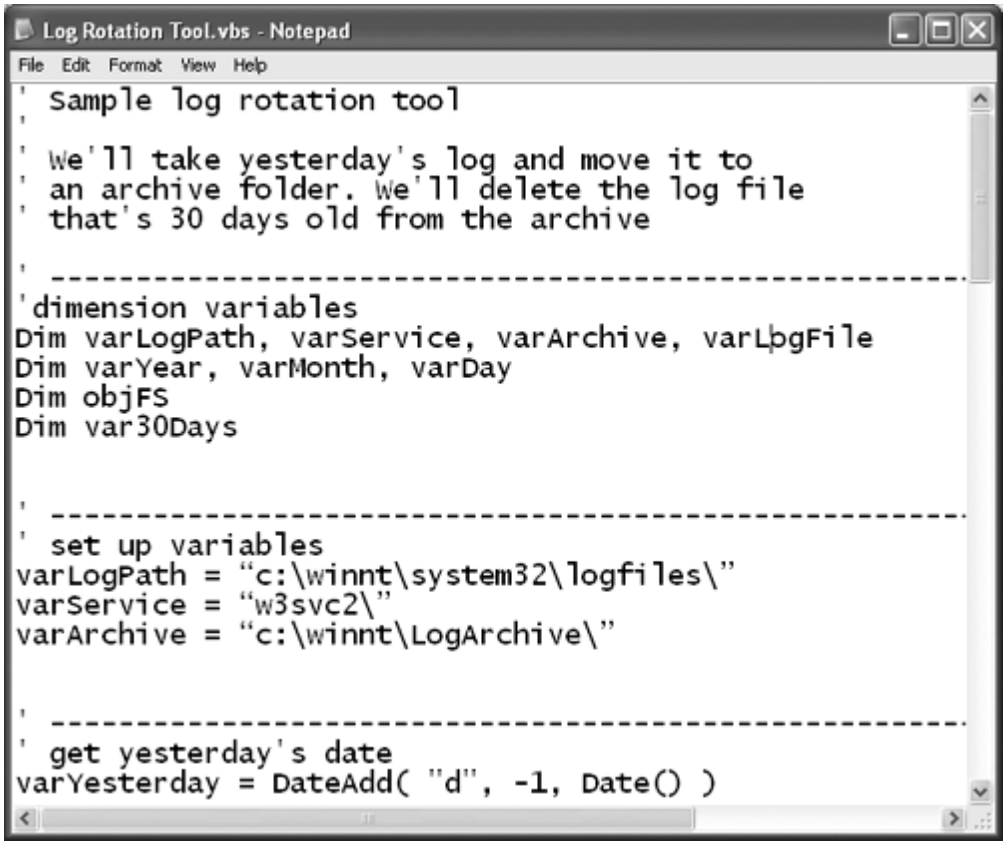
Notepad and Script Editors

When it comes time to write your scripts, you'll probably take the path of many administrators before you and start with Notepad. It's free, easy to use, and did I mention that it's free?! But you'll probably come to a point where you realize that Notepad is making you work too hard, and it'll be time to look at some professional alternatives.

Bare Bones: Notepad

Notepad, shown in Figure 2.9, is a basic text editor that makes a passable script editor. The biggest problem with Notepad that you'll notice right away is a lack of line numbering. When you get a VBScript error, it'll refer to a specific line number. Notepad does have a "Go to line number" feature that lets you type in the offending line number and jump straight to it, but it isn't as satisfying as if Notepad displayed a line number on every line of text.

Figure 2.9. Notepad as a script editor



Notepad also lacks any kind of color-coding, which can make scripting much easier, especially for long scripts.

A Step Up: Programmer's File Editor

Programmer's File Editor, or PFE, is a decent step up from Notepad. As shown in Figure 2.10, PFE can be configured to show line numbers on each line, making it easy to zip straight to the line of code that's causing errors.

Figure 2.10. Programmer's File Editor

Writing Your First Script

Because I don't expect you to plop down money to start scripting, I'm going to assume that you're either using Notepad or PFE as your script editor. I do highly recommend that you at least get PFE, because it's free and provides the all-important line numbering capability to your scripts. If you've decided to purchase another script editor, great! You shouldn't have any problems following along.

For your first script, I've selected a sample that will tell you which user or users, if any, have a particular file open through a shared folder on a file server. This can be a handy tool to have when you're trying to get to a file that's partially locked because someone else has it open. Listing 2.1 shows the complete script, which you can type into a text file and save as WhoHas.vbs.

Listing 2.1. WhoHas.vbs. **Displays the user or users who have a file open.**

```
' first, get the server name we want to work with

varServer = InputBox ("Server name to check")


' get the local path of the file to check

varFile= 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 objFS = GetObject("WinNT://" & varServer & "/lamanserver,fileservice")


' scan through the open resources until we

' locate the file we want

varFoundNone = True


' use a FOR...EACH loop to walk through the

' open resources

For Each objRes in objFS.Resources


     ' does this resource match the one we're looking for?

     If objRes.Path = varFile then


          ' we found the file - show who's got it

          varFoundNone = False
```

Running Your First Script

Double-clicking WhoHas.vbs should execute it in WScript. First, you'll be asked which server you want to connect to. Provide the server name, making sure you have administrative permissions on that server (the script will be using your user credentials to access the server). Next, provide the complete path and filename of the file you want to check.

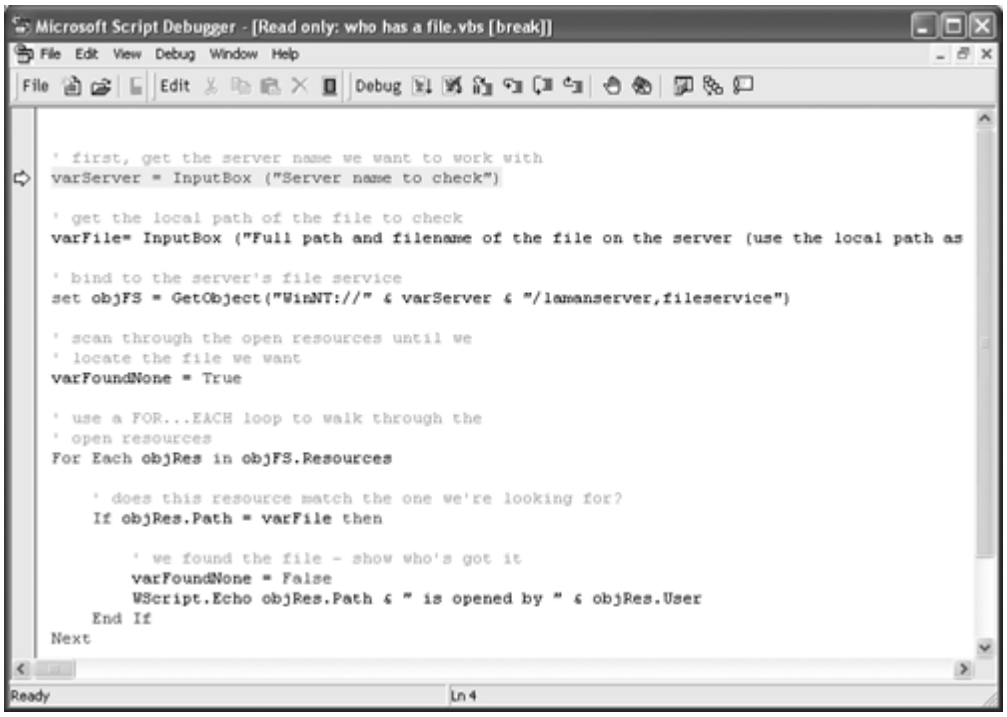
For example, suppose a folder on the server, named D:\Shares\Sales, is shared as Sales. A user is accessing a file named \\Server\Sales\SalesGoals.doc, and you want to find out which user it is. You'd type D:\Shares\Sales\SalesGoals.doc, because that's the server-local path to the file.

When you click OK…whoops! There's an error on line 10 (or another line, depending on how you typed the script)! That's not good! Looks like you're ready to start debugging your first script.

Debugging Your First Script

Microsoft offers a free script debugger from www.Microsoft.com/scripting. After you download and install the debugger, it will be available for the scripts that you write. In Figure 2.17, I've started the debugger, loaded WhoHas.vbs, and started execution. As you can see, the first line of code is highlighted, meaning the debugger is waiting to execute that code (it automatically skipped the very first line of text, which is just a comment line).

Figure 2.17. Debugging WhoHas.vbs



At this point, I can press F8 to execute just the highlighted line of text. Doing so displays an input box requesting the server name; after I provide that and click OK, the debugger jumps to the next line of text. A distinct disadvantage of the Microsoft Debugger is its lack of access to variable contents; in the VbsEdit debugger, I could hover my mouse over the varServer variable to verify that it contains whatever server name I had typed.

I can keep pressing F8 to execute each line of code, one at a time, until I run into the error again—as I do on line 10. Time to look at line 10 more carefully.

The problem, in fact, is the word lamanserver. It should be lanmanserver, with an "n" after the initial "la." Correcting that lets the script continue normally.

Often times, the debugger is the best way to see what "path" VBScript is taking. For example, your script might be behaving unexpectedly because you entered an incorrect logical comparison, perhaps typing ">" instead of "<" in a numeric comparison. Those types of errors won't necessarily generate errors—at least, not ones you'd be able to track down easily—but using the debugger can let you "walk" through script one line at a time as it executes and spot the location where the script's logic begins to go wrong.

Review

VBScript is easy to execute, because WScript.exe has been included with every major release of Windows since Windows 98. And, in Windows 2000 and later, WScript is even under Windows File Protection, ensuring that your users can't accidentally delete it. After you've taken precautions to ensure that only your scripts will execute (something I'll address in Chapter 28), you'll be ready to run!

Editing scripts can be a bit less satisfying. Windows doesn't come with any built-in tools specifically for editing scripts, and Notepad is a poor substitute. An advanced text editor like Programmer's File Editor makes things easier, and you can acquire some well-designed editors designed specifically for VBScript. Unfortunately, the most powerful script editors don't compare with the convenience and flexibility of professional software development tools like Visual Studio. On the other hand, your scripts probably won't ever be as complicated as most software development projects, so the extra convenience and flexibility isn't required.

COMING UP

Now that you know everything you need to create and execute scripts, you're ready to start learning how they work. Chapter 3 will cover the various components of a script, and introduce some sample scripts for you to review. Chapter 4 will let you practice everything you've learned in Part I by designing a script from scratch.

Chapter 3. The Components of a Script

IN THIS CHAPTER

When you started using Windows, you had to learn all kinds of new terms, like minimize and mouse. Scripting has a language all its own, too, and in this chapter, you'll learn the anatomy of a script and what some of those special terms mean.

Every good book has a structure. This book, for example, includes an introduction and some introductory chapters. Most of the book is taken up with explanatory chapters and examples. There's an appendix, an index, and a table of contents. All of these elements work together to make the book useful for a variety of purposes, including learning, referencing, and so forth.

Scripts have a structure, too. The main body of the script is a bit like its table of contents, organizing what the script will do. Functions and subroutines are the chapters of the book and perform the actual work. Finally, there are comments and documentation, which act as an index and help provide cross-references and meaning to the actual script code.

Do you really need to know these things to pump out a useful administrative script? Not at all. In fact, if your goal is to start programming as quickly as possible, skip ahead to the next chapter. However, understanding the structure of a good script can help make scripting easier, make your scripts more useful, and save you a lot of time and effort in the long run.

A Typical VBScript

Listing 3.1 shows the example script I'll be using in this chapter. This is actually a sneak preview; you'll see this script again in various forms in the next chapter. For now, don't worry much about what the script does or how it works; instead, just focus on what it looks like.

Listing 3.1. LoginScript.vbs**. This is the sample I'll be using throughout this chapter.**

```
'Display Message

MsgBox "Welcome to BrainCore.Net. You are now logged on."


'Map N: Drive

If IsMemberOf("Domain Users") Then

 MapDrive("N:","\\Server\Users")

End If


'Map S: Drive

If IsMemberOf("Research") Then

 MapDrive("R:","\\Server2\Research")

End If


'Map R: Drive

If IsMemberOf("Sales") Then

 MapDrive("S:","\\Server2\SalesDocs")

End If


'Get IP address

sIP = GetIP()


'Figure out 3rd octet

iFirstDot = InStr(0,sIP,".")

iSecondDot = InStr(iFirstDot+1,sIP,".")

iThirdDot = InStr(iSecondDot+1,sIP,".")

sThirdOctet = Mid(sIP, iSecondDot+1, _

 Len(sIP)-iThirdDot)
```

Functions

Functions are one of the workhorses of any script. They perform operations of some kind, and return some kind of result back to the main script. For example, VBScript has a built-in function called Date() that simply returns the current date.

There are built-in functions and custom functions that you write. The only difference between them, of course, is that Microsoft wrote the built-in functions and you write your custom ones. The sample login script has a couple of custom functions.

```
Function GetIP()
 Set myObj = GetObject("winmgmts:" & _
  "{impersonationLevel=impersonate}" & _
  "!//localhost".ExecQuery _
  ("select IPAddress from " & _
  "Win32_NetworkingAdapterConfiguration" & _
  " where IPEnabled=TRUE")

 'Go through the addresses
 For Each IPAddress in myObj
  If IPAddress.IPAddress(0) <> "0.0.0.0" Then
   LocalIP = IPAddress.IPAddress(0)
   Exit For
  End If
 Next
 GetIP = LocalIP
End Function

Function IsMemberOf(sGroupName)
  Set oNetwork = CreateObject("WScript.Network")
  sDomain = oNetwork.UserDomain
  sUser = oNetwork.UserName
  bIsMember = False
  Set oUser = GetObject("WinNT://" & sDomain & "/" & _
   sUser & ",user")
  For Each oGroup In oUser.Groups
    If oGroup.Name = sGroupName Then
```

Subroutines

The sample script has two custom subroutines, too. These are just like functions, except that they just do something; they don't return any result afterwards.

```
Sub MapDrive(sLetter, sUNC)

 Set oNet = WScript.CreateObject("WScript.Network")

 oNet.MapNetworkDrive sLetter, sUNC)

End Sub
```

```
Sub MapPrinter(sUNC)

 Set oNet = WScript.CreateObject("WScript.Network")

 oNet.AddWindowsPrinterConnection sUNC

 oNet.SetDefaultPrinter sUNC

End Sub
```

These subroutines are declared with the word Sub, followed by the name of the subroutine. Like a function, these two subroutines each accept some input parameters. Unlike a function, they never set their name to some value, which is why they don't return a value.

VBScript has intrinsic (built-in) subroutines, only they're called statements. A simple statement, like Beep, simply makes the computer beep.

Subroutines serve the same purpose as a function: Although mapping a drive or a printer obviously isn't difficult (taking only two or three lines of code), there's no reason I should have to type those lines of code over and over. Encapsulating the functionality into a subroutine means I can reuse the code repeatedly in one script, and easily paste it into other scripts, saving myself work.

Main Script

The main script performs a good bit of work. Here it is in its entirety.

```
'Display Message

MsgBox "Welcome to BrainCore.Net. You are now logged on."


'Map N: Drive

If IsMemberOf("Domain Users") Then

 MapDrive("N:","\\Server\Users")

End If


'Map S: Drive

If IsMemberOf("Research") Then

 MapDrive("R:","\\Server2\Research")

End If


'Map R: Drive

If IsMemberOf("Sales") Then

 MapDrive("S:","\\Server2\SalesDocs")

End If


'Get IP address

sIP = GetIP()


'Figure out 3rd octet

iFirstDot = InStr(0,sIP,".")

iSecondDot = InStr(iFirstDot+1,sIP,".")

iThirdDot = InStr(iSecondDot+1,sIP,".")

sThirdOctet = Mid(sIP, iSecondDot+1, _

 Len(sIP)-iThirdDot)


'Map printer based on octet

Select Case sThirdOctet

 Case "100"
```

Comments and Documentation

Documenting your scripts is always a very good idea. After all, the script makes perfect sense now, but will you be able to figure out what it's doing a year, or even a couple of months, from now?

For example, examine the script in Listing 3.2. See if you can figure out what the various portions of the script are doing.

Listing 3.2. AddUsersFromXLS.vbs**. This script auto-creates users from an Excel spreadsheet.**

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

oCN.Open "Excel"

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

Set oDomain = GetObject("WinNT://NT4PDC")

Set oFSO = CreateObject("Scripting.FileSystemObject")

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

sHomePath = "\\iridis1\c$\users\"

Do Until oRS.EOF

sUserID = oRS("UserID")

sFullName = oRS("FullName")

sDescription = oRS("Description")

sHomeDir = oRS("HomeDirectory")

sGroups = oRS("Groups")

sDialIn = oRS("DialIn")

sPassword = Left(sUserID,2) & DatePart("n",Time) & _

DatePart("y",Date) & DatePart("s",Time)

Set oUserAcct = oDomain.Create("user",sUserID)

oUserAcct.SetPassword sPassword

oUserAcct.FullName = sFullName

oUserAcct.Description = sDescription

oUserAcct.HomeDirectory = sHomeDir

If sDialIn = "Y" Then

oUserAcct.RasPermissions = 9

Else

oUserAcct.RasPermissions = 1

End If

oUserAcct.SetInfo
```

Review

In this chapter, I tried to illustrate some of the different components of a good script. You've seen what functions, statements, and subroutines look like, how a main script ties them all together, and how comments and documentation make them easier to read and maintain in the future. Keep all these new concepts in mind as you move through the rest of the book. Try to spot intrinsic functions and custom ones, and watch for comment lines and other types of code documentation. Try to use these standards in your own scripts, and you'll find yourself becoming more efficient and more capable very quickly.

COMING UP

You're almost ready to start learning VBScript, but you need to learn how to design your scripts first. In the next chapter, I'll show you my simple methodology for designing scripts that will let you write powerful administrative scripts without becoming a career programmer.

Chapter 4. Designing a Script

IN THIS CHAPTER

Believe it or not, the toughest part about scripting isn't the language or programming objects or anything technology related. It's in how you design your scripts and get them to do what you want. I'll share tips and techniques that'll make scripting easier and more approachable.

Suppose you want to do a tune-up on your car, and you don't want to hire a mechanic to do the job for you. It's easy enough to run down to the hardware store and acquire the necessary tools, and you can even buy some books that explain how to use those tools. If you're like me, though, none of that will help you get the tune-up done. Where do you start? What should it look like? Which tools do you use, and when?

I've found that's how many administrators feel about scripting. Sure, the VBScript documentation is available, and there are plenty of examples on the Web. But where do you start when it comes time to write your own scripts? Much of this book will be focused on the tools, like VBScript and programming objects, that you'll need to do the job. In this chapter, I want to share some of the tips and techniques that I use to actually get started in designing a new script.

It'll be easier to see how my design process works with a meaningful example. Because login scripts are a popular administrative use of VBScript, I'll use a login script as an example. I want to write a login script that maps three network drives based on the users' group membership, and then maps a printer based on the user's DHCP-assigned IP address. That way, I can assign a printer that's local to wherever the user logged on. I also want to display a welcome message, and I want the script to run on Windows NT Workstation 4.0 (Service Pack 6a or later), Windows XP Professional (Service Pack 1 or later), and Windows 2000 Professional (Service Pack 3 or later).

Creating a Task List

The first thing to do in the design process is to create a task list. This is essentially an English-language version of the script you plan to write. In the list, you must break down the various things you want the script to perform in as much detail as possible. I often go through several iterations of the task list, adding a bit more detail each time through. Listing 4.1 shows what my first pass might look like.

Listing 4.1. Login script task list**. Your first task list should just summarize what you want the script to do.**

```
Login script task list



Display a logon welcome message.

Map the N: drive based on group membership.

Map the R: drive based on group membership.

Map the S: drive based on group membership.

Map a printer and make it the default. Base the printer selection on the user's physical
location at the time.
```

NOTE

Programmers call this kind of a task list pseudocode, because it sort of looks like programming code but isn't. It's a great way to lay out what a script is supposed to do without having to look up the exact VBScript syntax of every command. Plus, you can throw around phrases like, "I just finished pseudocoding that script, and boy was it tough," and you'll impress the software developers in your company.

After you've got your first task list completed, look at what detail might be missing. For example, "based on group membership" is vague. What specific parameters will be used to determine where the N: drive is mapped? Will the N: drive always be mapped, or will it be mapped only if the user is in one or more specific groups? Pretend you're explaining how the script will work to the least technical person you know, and add the level of detail they'd need to understand what the script should do. Listing 4.2 shows a second, more detailed attempt.

Listing 4.2. Login script task list II**. Adding detail makes the task list more useful.**

```
Login script task list v2



Display a logon welcome message:

"Welcome to BrainCore.Net. You are now logged on."



Map the N: drive:

If the user is a member of the Domain Users group.

Map the drive to \\Server\Users.
```

Selecting the Appropriate Tools

Now comes what is truly the most difficult part of administrative scripting: selecting the right tools. You know what you want your script to do, and you know VBScript can do it (or you at least hope it can), so you just need to figure out how to make it work.

Software developers do this all the time. Typically, they know so much about the tools they have to work with, though, that they select the right ones without even thinking about it. As an administrator, I'm more likely to have to do some research first.

Looking at the task list, there are really six types of tasks I need the computer to perform:

1. Displaying a message
2. Mapping a drive
3. Checking group membership
4. Mapping a printer
5. Getting the local IP address
6. Getting the third octet from the IP address

I'll show you how I research each of these tasks to figure out how they can be accomplished.

TIP

The Appendix of this book is a quick script reference. It's designed to list common administrative tasks and briefly describe what VBScript tools you can use to accomplish them. This reference should make it easier for you to figure out which tools to use when you write your own scripts.

Displaying a Message

I always start my research in the VBScript documentation. You can find it online at www.microsoft.com/scripting, and there's even a downloadable version that you can use offline. You can also find the documentation in the MSDN Library. That's available online at http://msdn.microsoft.com/library, or you can receive it on CD or DVD as part of a yearly subscription. In either case, I find an offline version of the docs to be more convenient.

TIP

At the very least, download the offline VBScript documentation. Go to www.microsoft.com/scripting and look for the appropriate link. The actual download URL changes from time to time, so you're better off starting at the main scripting page and locating the link.

When I need to search the VBScript documentation, I usually start with the alphabetical list of available functions and commands. That's just an easy way for me to scan through the docs and spot likely looking tools. In this case, down in the M section of the function list, I ran across MsgBox. Even if you know nothing about VBScript, MsgBox certainly sounds as if it displays a message box. Looking into the details of the function, I see that it does, in fact

Creating Modules to Perform Tasks

After you've got your task list nailed down, and you've figured out how to perform each of the tasks in script, you can start designing the modules of the script. I often have to spend a lot of time figuring out how to do things like look up IP addresses or connect to domain controllers; after I've spent that time, I don't ever want to have to do it again. In other words, I want to modularize my scripts, so that difficult or commonly used tasks can be easily cut and pasted into future scripts.

VBScript provides a way for you to write your own functions and statements, making it easy to modularize your code. Most of the time, the tasks your script accomplishes—in this case, mapping drives, getting IP addresses, and so forth—can be easily written as functions and subroutines, which can be easily cut and pasted into future scripts.

For a quick overview of functions and statements, see "What Are Functions?" in Chapter 5. You can see how they fit into a script in "Functions" and "Subroutines" in Chapter 3. Finally, I provide more detail on modular script programming in Chapter 25.

Probably the best way to see how these tasks can be modularized is with an example of the completed login script.

➤➤ The Login Script

Listing 4.5 shows what the various functions for the logon script might look like, and also shows how the main script might be written to call on each of these functions.

NOTE

Don't worry for now about how this script actually works. You'll be seeing all of these features again in later chapters, where I'll provide explanations that are more detailed. For now, just focus on how the various things are broken into modules that make them easier to reuse throughout the main script.

Listing 4.5. LoginScript.vbs**. This script includes a main script as well as functions, making a modular script.**

```
'Display Message

MsgBox "Welcome to BrainCore.Net. You are now logged on."


'Map N: Drive

If IsMemberOf("Domain Users") Then

 MapDrive("N:","\\Server\Users")

End If


'Map S: Drive

If IsMemberOf("Research") Then

 MapDrive("R:","\\Server2\Research")

End If


'Map R: Drive
```

Validating User Input

This example logon script doesn't have any user input, but some of your scripts may. For example, you might write a script that asks for a server name, and then does some operation on that server. Any time you're asking for user input, you need to validate that input to make sure it's within the range that you expected.

For example, suppose you have a script that shuts down a remote server. You might have the script ask for the server name, and then ask for a shutdown delay in seconds. After accepting that input from the script's user (who might even be you), the script should check to make sure the server name was valid (perhaps it must start with two backslashes), and that the delay was within an acceptable range (maybe 5–30 seconds).

You can generally use If…Then constructs to validate user input. Why bother? Validation ensures that your scripts are working with proper input, and can help prevent the scripts from running into errors or performing unexpected actions.

Chapter 10 introduces If…Then, under "Conditional Execution."

If users provide incorrect or unexpected input, your script can display an error message and end, or even give users another chance to enter the necessary information.

TIP

Plan to add user validation to all scripts that accept input from a user. The examples in this book don't always include input validation; I've deliberately left it out in many cases to help focus on what the script is supposed to accomplish. Scripts used in the real world, however, should always validate user input.

Planning for Errors

Errors occur. There are actually a few different types of errors, with specific ways of dealing with each.

- Syntax errors. These are simple typos that you introduce when writing a script. You'll generally catch these when you test your scripts.
- Logic errors. These design flaws make the script behave unexpectedly or incorrectly. Again, these are usually your fault, and you'll find them as you test the script.
- Conditional errors. These errors occur because something in the script's operating environment was other than what you planned for when you wrote the script. For example, a domain controller might be unavailable, or a user may have typed a server name that doesn't exist.

Syntax and logic errors often crop up in scripting, and you'll find them as you test and debug your scripts. Conditional errors, however, are generally beyond your control. Your scripts should try to anticipate these errors, however, and handle them gracefully. For example, suppose you're using the WScript.Network object to map a network drive, and the server happens to be unavailable at the time. The basic script might look like this:

```
Set oNet = WScript.CreateObject("Wscript.Network")

oNet.MapNetworkDrive "S:", "\\Server2\SalesDocs"
```

If Server2 isn't available, the script will crash when executing the second line of code. That means the script will display an error message and won't execute anything else in the script. You can make the script a bit more resilient by anticipating the problem and adding error-handling code.

```
Set oNet = WScript.CreateObject("Wscript.Network")


On Error Resume Next

oNet.MapNetworkDrive "S:", "\\Server2\SalesDocs"

If Err <> 0 Then

 MsgBox "Server2 was unavailable; your S: drive was not mapped."

End If

On Error Goto 0
```

This modified script starts out by telling VBScript, "Look, if an error occurs, it's OK, I'll handle it. You just resume execution with the next line of script." That's done by On Error Resume Next.

After trying to map the drive, an If…Then construct checks the value of the special variable Err. If it's zero, the drive was mapped. If not, a friendlier error message is displayed to the user letting him know something went wrong. Finally, error checking is turned off with On Error Goto 0. From then on, errors will result in a VBScript error message and the script will stop executing.

Creating Script Libraries

After you've created some useful functions, you can save them into a script library. That's nothing any fancier than a collection of useful scriptlets, which you can reuse in various scripts that you write. For example, you might pull out all of the functions and subroutines from the logon script you wrote, saving them into a separate file. That'll make it easier to reuse those useful bits of code later in the future.

By carefully modularizing your code, you'll quickly build a library of useful scripts, making it easier to write new scripts in the future.

Review

In this chapter, I tried to provide you with a look at how I go about designing and writing scripts. I don't just sit down and start typing; instead, I create a list of tasks I want the script to accomplish, and then I try to do some research and find out exactly how to perform each of those tasks in a script. The VBScript documentation, Google, and other Web resources are useful for finding examples and information, and before long I usually find everything I need to know. Next, I try to modularize the script, so that I can reuse my hard-earned information in other scripts that I might write in the future.

If you approach script design and development with this methodical approach, you won't need to be an expert developer to write great scripts. You can build on the work of those who came before you, and quickly start writing scripts that are useful in your environment.

COMING UP

You're ready to start learning VBScript, and your crash course begins in the next chapter. Don't worry; you're not going to be turned into a programmer! Instead, you'll be learning just enough VBScript to have some powerful tools at your disposal as you start scripting.

Part II: VBScript Tutorial

Chapter 5. Functions, Objects, Variables, and More

IN THIS CHAPTER

You've learned how to design a script, but you're not quite ready to write one. First, you have to learn the basics of VBScript, and this chapter begins your crash course.

Scripting is, of course, a form of computer programming, and computer programming is all about telling a computer what to do. Before you can start ordering the computer around, though, you need to learn to speak a language that it understands. VBScript is one such language, and in this chapter, I'll introduce you to the VBScript syntax, or language.

Almost all computer programming languages, including VBScript, have a few things in common.

- They have built-in commands that tell the computer to perform certain tasks or calculate certain kinds of information.
- They have a means for tracking temporary information, such as data entered by a user or collected during some calculation.
- Windows-based programming languages generally have a means for interacting with objects, because objects form the basis of Windows' functionality.

NOTE

The capability to interact with objects is not the same thing as being an object-oriented programming language. Although the concepts and benefits of object-oriented programming are beyond the scope of this book, suffice to say that VBScript isn't object oriented, despite its capability to interact with objects created in other languages.

VBScript implements these common programming elements through variables, functions, and statements, and through an object interface.

- Variables act as storage areas for different types of data.
- Functions are VBScript's way of performing calculations or tasks and providing you with the results; statements simply perform tasks. You can even create your own functions and statements to customize VBScript's capabilities.
- VBScript includes a complete object interface based on Microsoft's Component Object Model, or COM.

In this chapter, you'll learn how to use each of these elements within scripts.

NOTE

I've never liked programming books that provide short, useless snippets of script as examples, even as early in the book as you are right now. Most of the examples you'll see in this and subsequent chapters are fully functioning scripts

What Are Variables?

Variables are temporary storage areas for data. You may even remember them from algebra: x + 5 = 10, solve for variable x. Of course, in those situations, x wasn't really a variable, because it always equaled some fixed amount when you solved the equation. In scripting, variables can change their contents many times.

➤➤ Sample Script

Listing 5.1 shows a sample script. It's a fully functional script that will connect to a domain, locate any inactive user accounts, and disable them.

NOTE

Once more, I'm showing you a script that uses some advanced features. This lets me show you functional, useful scripts rather than dumbed-down examples, but for now I'm just going to explain the bits that are important for this chapter. I promise you'll get to the rest later!

Listing 5.1. DisableUser.vbs**. We'll use this script as a running example throughout this chapter.**

```
Dim sTheDate

Dim oUserObj

Dim oObject

Dim oGroupObj

Dim iFlags

Dim iDiff

Dim sResult

Dim sName

Const UF_ACCOUNTDISABLE = &H0002


' Constant for Log file path

 Const sLogFile = "C:\UserMgr1.txt"


' Point to Object containing users to check

Set oGroupObj = GetObject("WinNT://MYDOMAINCONTROLLER/Users")


On Error Resume Next

For Each oObject in oGroupObj.Members


 ' Find all User Objects Within Domain Users group

 ' (ignore machine accounts)
```

What Are Functions?

Functions are a way of performing a task and getting something back. For example, VBScript has a function named Date(), which simply looks up and provides the current date according to your computer's internal clock. Functions are used to perform special calculations, retrieve information, look up information, convert data types, manipulate data, and much more.

Input Parameters

Functions may include one or more input parameters, which give the function something to work with and usually are a major part of the function's output. Not all functions need input, however. For example, the Date() function doesn't need any input parameters to function; it knows how to look up the date without any help from you.

Other functions may require multiple input parameters. For example, the InStr() function is used to locate a particular character within a string. Here's how it works.

```
Dim sVar

Dim iResult

sVar = "Hello!"

iResult = InStr(1, sVar, "l")
```

After running this short script, iResult will contain the value 3, meaning the Instr() function located the letter l at the third position within the variable sVar. InStr() requires three input parameters:

1. The character position where the search should start
2. The string in which to search
3. The string to search for

NOTE

Of course, I haven't necessarily memorized InStr()'s input parameters. I looked them up in the VBScript documentation. After you use a function a few times in scripts, you'll remember its parameters without looking them up, but I don't use InStr() very often so I always refer to the documentation to see in which order the parameters should be.

Now that you know what a function looks like, refer to this section of the DisableUsers sample script and see if you can spot the functions (I've boldfaced them to make it easy).

```
' get last login date

  sTheDate = UserObj.get("LastLogin")

  sTheDate = Left(sTheDate,8)

  sTheDate = CDate(sTheDate)
```

What Are Statements and Subroutines?

Here's where VBScript's terminology gets a bit complicated, and for no good reason: Aside from the terms themselves, statements and subroutines are actually quite straightforward. A statement is an intrinsic command that accepts zero or more parameters and returns no value. A subroutine is simply a custom statement that you write yourself. Intrinsic and custom functions are both called functions; why custom statements are called subroutines (or subs for short) is a mystery from the depths of VBScript's past.

Functions, without the Output

Statements (and subroutines) always perform some kind of task. Unlike a function, statements cannot return a value to your script, so they just perform a task. One of the simplest VBScript statements is Beep, which simply makes the computer beep. It takes no parameters, returns no value, and performs one task. Another simple statement, End, tells VBScript to stop running your script immediately. Pretty simple!

➤➤ A Custom Subroutine

You may want to create custom routines that perform a task but return no value. For example, suppose you're writing a script and you want your computer to beep a lot when it encounters some specific condition, such as an error or a full hard disk. You could just list multiple Beep statements in a series, but it would be more efficient to use a custom subroutine. You could program the subroutine to accept an input parameter describing how many times to beep. Listing 5.3 shows an example.

Listing 5.3. MultiBeep Subroutine. **This subroutine can be used to make the computer beep a specified number of times.**

```
Sub MultiBeep(iTimes)

 Dim iTemp

 For iTemp = 1 To iTimes

  Beep

 Next

End Sub
```

You can use the MultiBeep subroutine from anywhere in the main portion of your script. For example, to make the computer beep five times, you'd use MultiBeep(5).

➤➤ A Custom Subroutine—Explained

The MultiBeep subroutine actually uses several VBScript commands I haven't introduced yet, but I'll focus for now on the parts that define the subroutine. First, all subroutines include a Sub and an End Sub statement, in much the same way that custom functions use Function and End Function:

```
Sub MultiBeep(iTimes)

End Sub
```

As with a custom function, note that the parameters are defined in the Sub statement. In this case, the variable iTimes represents the number of times the computer should beep.

What Are Objects?

I've already made a big deal about how VBScript lets you access operating system functionality because VBScript is object based, and Windows exposes much of its functionality through objects. So you may be wondering, "What the heck is an object?"

Bear with me for the 10-second synopsis. You may have heard of COM or COM+, two versions of Microsoft's Component Object Model. The whole idea behind COM is that software developers can package their code in a way that makes it easily accessible to other applications. For example, suppose some poor developer spent a few years developing a cool way to interact with e-mail systems. If the developer wrote that code according to the rules of COM, every other developer—or scripter—would be able to take advantage of that e-mail interaction. In fact, a bunch of developers did exactly that! You may have heard of Microsoft's Mail Application Programming Interface, or MAPI. It's what Microsoft Outlook uses to access an Exchange Server, for example. MAPI is an example of COM in action; any programmer—including you—can use MAPI to access a mail server, because MAPI is written to the COM standard.

Therefore, an object is simply a piece of software that's written to the COM standard. VBScript can use most objects that are written to the COM standard; most of Windows' functionality is written to the COM standard, and that's what makes VBScript so powerful.

Properties

Most software requires some kind of configuration to use it, and COM objects are no exception. You configure an object by setting its properties. Properties are simply a means of customizing an object's behavior. For example, an e-mail object might have properties for setting the mail server name, setting the user's name and password, and so forth.

Properties can also provide information to your script. A mail object might include a property that tells you how many new messages are available or how much space is left in the mailbox.

In your scripts, you'll generally use a variable name to represent an object. I use variable names that start with the letter o, so that I know the variable is really an object reference. To refer to an object's properties, simply list the property name after the object name. For example, suppose you have a mail object referenced by the variable oMail. To set the mail server name, you might use oMail.ServerName = "mail.braincore.net". The period in between the object variable and the property name helps VBScript distinguish between the two.

Methods

You already know that functions, statements, and subroutines exist in VBScript. Objects have functions, statements, and subroutines too, but they're called methods. Suppose your fictional mail object provided a statement named GetMail, which retrieved mail from the mail server. You could then simply include oMail.GetMail in your script to activate the statement and retrieve the mail.

Like functions and statements, some methods accept input parameters. For example, oMail.GetMail(1) might retrieve the first message in the mailbox. Other methods might work as functions sMessage = oMail.GetMail(2) might retrieve the second message and store the message body in the variable sMessage.

How do you know what methods an object supports? Check the documentation. Also, I'll introduce you to several useful objects in Chapters 11 and 12.

Collections

Sometimes, programmers create objects that represent a hierarchy of real-world data. One common hierarchy that you're probably familiar with is the file system on a computer: It's a tree of folders and files. If you wanted to manipulate the file system in a script, you'd need an object that represented that hierarchy of folders and files.

Review

You've started learning how VBScript works in this chapter. In fact, you've learned about the three main parts of any script: functions and subroutines (which you now know aren't really that different from one another), objects, and variables.

Variables act as temporary storage areas for your data and allow your scripts to change their behavior and manipulate data. VBScript's built-in functions and statements provide the actual functionality of the language, whereas your own functions and subroutines extend VBScript's power to perform custom tasks.

Finally, objects represent the functionality of the Windows operating system and its many features and capabilities. Objects have properties, which govern their behavior, and methods, which perform actions. Administrative scripting is all about using VBScript functions and statements to tie together operating system objects. For example, you might use a file system object to manipulate files and folders or use the WMI objects to manipulate the registry.

COMING UP

In the next chapter, you'll learn how VBScript accepts input and displays messages, enabling you to create interactive scripts. Chapters 7 through 9 show you how to manipulate the data that your scripts work with. If you're anxious to start working with objects, jump to Chapter 11, which introduces some of VBScript's own built-in objects.

Chapter 6. Input and Output

IN THIS CHAPTER

Getting and displaying information is an essential function for most scripts. In this chapter, you'll learn the various ways that VBScript provides for this type of user interaction.

It's rare to need a script that doesn't involve some form of user interaction. At the very least, you might need to display some kind of completion message as an indication that your script has finished running. Sometimes, you'll need more complex interaction, such as the ability to ask Yes or No questions, get server names and other information, and so forth.

VBScript has very limited interactive capabilities. If you're expecting to create even simple dialog boxes like you've seen Visual Basic programmers do, forget about it: VBScript doesn't provide a dialog builder and doesn't provide any means for programmatically creating dialog boxes. If you need a custom user interface, you need to upgrade to a full-fledged programming environment like Visual Studio. VBScript's capabilities for interaction are limited to basic choices, simple messages, and one-line text input. However, in an administrative script, that's often all you'll need.

NOTE

The script examples in this chapter won't be full administrative scripts. Instead, I'll provide snippets that you can easily cut and paste into your own scripts whenever you need to display a message, ask for user input, and so forth.

Displaying Messages

VBScript displays messages using the Windows standard message box, which is a short dialog box that has a few display options to customize its behavior and appearance. VBScript exposes this functionality through the MsgBox statement and the MsgBox() function.

The MsgBox Statement and Function

MsgBox is one of the few VBScript elements that can act as both a statement and a function. As a statement, MsgBox just displays a message box to your specifications. As a function, however, MsgBox can act as a form of user input, allowing simple Yes/No choices that can affect the behavior of your scripts.

The basic MsgBox statement accepts up to three parameters: a message, a numeric value designating which system icons or buttons should be displayed, and a message box title. It looks something like this:

```
MsgBox "The script has finished running.", _
 1, "Notice"
```

This command will display a message box that contains the text "The script has finished running." The box will include an OK button and a Cancel button, and the title of the box will contain "Notice." If you don't care about the title of the dialog or the buttons it displays, you can take a shortcut and just include your message.

```
MsgBox "The script has finished running." & _
 " Please check the server for the new user accounts."
```

The default message box title will be displayed, and the default button configuration—an OK button and a Cancel button with no icon—will be displayed.

Your scripts will look cooler, though, if you customize them a bit. For example, you might display an information icon on the message box, which helps cue the user that the message isn't an error or a question, but a simple informative message. You might also display just an OK button; a Cancel button doesn't really make sense because when the script is done, there's nothing left to cancel.

```
MsgBox "The script has finished running.", _
 64, "Thank you."
```

TIP

When you include a system icon, Windows will play any associated event sounds when your dialog box is displayed. This feature makes your script seem much more professional and integrated with the operating system.

That middle parameter—the number 64, in this case—controls the icons and buttons that display on the dialog box. Table 6.1 shows the options you have available, along with their corresponding values.You can pick from four classes of options.

1.

Asking for Input

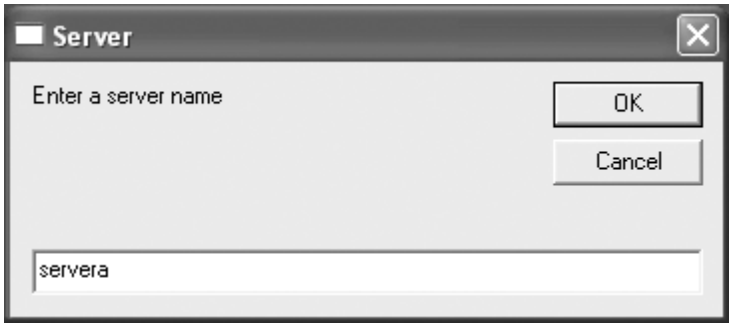
Although the MsgBox() function provides a way to collect simple Yes/No style input, you may need to collect more complex input, such as server names, user names, or other data. VBScript provides a way for users to input this type of string information.

Graphical Input

The InputBox() function displays a graphical input box with a title, a short message, a one-line text input box, and an OK and Cancel button. Whatever the user types is returned as the result of the function; if the user clicks Cancel or presses Esc on his keyboard, the function returns -1. Figure 6.4 shows what this quick sample looks like.

```
Dim vInput

vInput = InputBox("Enter a server name","Server")

MsgBox "You entered " & vInput
```

Figure 6.4. Collecting text input by using InputBox()



You should always test to see if the user clicked Cancel or pressed Esc.

```
Dim vInput

vInput = InputBox("Enter a server name","Server")

If vInput = -1 Or vInput = "" Then

 MsgBox "You cancelled."

Else

 MsgBox "You entered " & vInput

End If
```

This type of check prevents your script from trying, for example, to connect to a server named \\-1 when the user cancels the input box.

You can expand InputBox() slightly to provide a default entry. Users can accept your default by simply clicking OK or pressing Enter when the input box is displayed, or they can type their own input instead of your default. Here's how.

```
Dim vInput, vDefault
```

Command-Line Parameters as Input

Many of the administrative utilities you use every day are command-line utilities, such as ipconfig, ping, and tracert. These utilities can all perform different tasks, or different variations of a task, through the use of command-line parameters. For example, ipconfig /all displays IP configuration settings, whereas ipconfig /renew refreshes your computer's DHCP address. You also can write scripts that accept command-line parameters, giving you the ability to create flexible command-line utilities of your own.

NOTE

Command-line scripts usually execute under Cscript.exe, giving them the capability to produce command-line output. Keep in mind that you'll need to use WScript.Echo rather than MsgBox to produce the command-line output.

Running Command-Line Scripts

Because none of the script file extensions—VBS, VB, SCR, and so forth—are recognized as executable by the Windows command-line processor, you'll need to execute Cscript.exe directly. Tell Cscript which script file you want to execute, and then tack on any of the script's command-line parameters, followed by any Cscript parameters. For example:

```
Cscript.exe MyScript.vbs /option:yes /value:4 //B
```

This would execute the VBScript MyScript.vbs, passing it a parameter named option and one named "value," and telling Cscript to suppress any script error messages (see the sidebar, "Power Cscript.exe," later in this chapter for more on Cscript parameters).

Parsing Parameters

The scripting engine includes a built-in parameter-parsing object named WshNamed, which is designed to help your script accept named command-line parameters. Note that this object is also available to graphical scripts executing under WScript, although it's less common to see those scripts using command-line parameters. WshNamed is part of the WshArguments object, which provides top-level access to all command-line arguments passed to the script. I introduced you to objects in VBScript in "What Are Objects?" in Chapter 5.

Suppose you're writing a script that will display basic information about a remote computer. You want the script to accept a command-line parameter named Computer that will provide the computer name to check. You'll execute the script with something like the following:

```
Cscript.exe GetInfo.vbs /computer:server1
```

You want the script to run from the command line, so you'll display the output by using WScript.Echo instead of the MsgBox statement.

➤➤ Getting Remote Machine Information

Listing 6.1 shows what your script might look like.

Listing 6.1. GetInfo.vbs. **This script will retrieve basic information about a remote computer.**

Review

Scripts can be made more general-purpose when they're capable of collecting input and customizing their behavior based upon that input. You can have scripts connect to different servers, create user accounts, delete files, and perform hundreds of other actions when you're able to collect and evaluate user input when the script is run.

Using MsgBox allows you to display messages, even those with some basic formatting. You can also ask for simple Yes/No decisions from the user by using MsgBox() as a function. WScript.Echo provides text-output capabilities that work within a graphical WScript or command-line Cscript environment.

InputBox() allows you to collect text input from a graphical script, and you can use WScript.StdIn to collect text input within a command-line script. You can also use script command-line parameters to create scripts that work just like Windows' built-in command-line tools, complete with named parameters that customize the script's behavior.

COMING UP

In the next chapter, you'll start learning how to manipulate data in VBScript. I'll start with numeric data in Chapter 7, move on to string data in Chapter 8, and finish up with other types of data in Chapter 9.

Chapter 7. Manipulating Numbers

IN THIS CHAPTER

Most scripts involve math and numbers, and learning how to work with them is important. Other types of data manipulation, including strings and dates, build upon basic number manipulation, too, making this a core skill. Fortunately, VBScript makes numeric data easy to work with.

Incredibly, I almost became a professional software developer. It's what I wanted to do in high school, and my school even offered a two-year vocational course for programmers. I never pursued it, though, because I'm horrible with higher math. In school, everyone tells you that programmers have to really know their math. That's not true, of course. With VBScript in particular, you'll find that manipulating numeric data and performing even complex calculations is easy. Even better, you probably won't need more than basic math skills for administrative scripting.

Numbers in VBScript

VBScript considers any unquoted, non-date value to be a number. Issuing the statement MyVariable = 5, for example, will assign the numeric value 5 to the variable MyVariable. The one catch in VBScript is that there are actually different types of numbers.

- Any whole number—that is, a number with no decimal portion—is called an integer. The numbers 5, -6, 43,233, and -42 are all integers. VBScript integers can be anything from -32,768 to 32,767.
- VBScript also supports long integers, which are just big integers. They can be anything from -2,147,483,648 to 2,147,483,647.
- Numbers with a fractional value can be either singles or doubles. The only difference between them is in how large they can be. A single can be any numeric value from -3.4028235E+38 to -1.401298E-45, or from 3.4028235E+38 to 1.3401298E-45. In other words, a really big number. Sometimes, however, you may need an even larger number, which is where doubles come in. A double can be truly huge—as big as 1.79769313486231570E+308. I have no idea what you'd call a number like that other than humongous.
- VBScript also supports a currency number type. This has a maximum precision of four decimal places and has the added capability to properly recognize and format currencies based on the system's locale settings. That means you can properly display thousandths separators and decimal places according to the system configuration.

Now, as I mentioned in Chapter 5, you don't usually have to worry much about these different types of numbers, because VBScript does it for you. Variables in VBScript can hold any kind of data; if you try to put the number 64,555 into a variable, VBScript will just invisibly make the variable into a long integer. If you add .3 to it, VBScript will convert it into a single. The only time you'll need to worry about data types is if you want to perform some specialized function, like a currency operation, and then you'll need to explicitly convert the variable into the appropriate type—something I'll cover later in this chapter.

Basic Arithmetic

You use VBScript's mathematical operators for most basic math. VBScript's operators should look pretty familiar and self-explanatory:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Order of evaluation: ( )
- Exponentiation: ^ (usually located in the 6 key on your keyboard)

Normally, you just assign the results of such an operation to a variable or a statement like MsgBox, as in the following examples.

```
myVar = 1 + 2

MsgBox myVar

myVar = myVar + (myVar * .03)

MsgBox myVar

myVar = myVar^2

MsgBox myVar
```

VBScript evaluates expressions from left to right, performing exponentiation first, then multiplication and division, then addition and subtraction. Anything in parentheses is evaluated first, starting at the innermost level of nested parentheses. To put that in context, consider these similar-looking expressions, which all generate a different result due to the order of evaluation.

```
myVar = 1 * 2 + 1

MsgBox myVar

myVar = 1 + 1 * 2

MsgBox myVar

myVar = (1 + 1) * 2

MsgBox myVar

myVar = ((1 + (1 * 2))
```

Advanced Arithmetic

If you're getting heavy-duty with the math in a script, you may need to take advantage of some of VBScript's advanced math functions. These include

- Atn(): Arctangent
- Cos(): Cosine
- Sin(): Sine
- Tan(): Tangent
- Log(): Logarithm
- Sqr(): Square root
- Exp(): Returns e (the base of natural logarithm) raised to a power
- Randomize: Randomizes the system's random number generator
- Rnd(): Generates a random number

This random number business in particular deserves some explanation, because you may think that'd be a much better way to come up with values for a password. It can be, provided you thoroughly understand how randomness works inside a computer.

First, never forget that computers are giant calculating devices. There's nothing remotely random about anything that goes on inside a computer. As a result, no computer can generate a truly random number without special hardware that's designed to do so.

The Rnd() function returns a value less than 1, but greater than or equal to zero. You can pass a parameter to Rnd() to control its behavior.

- A number less than zero, such as Rnd(-2), will return the exact same number every time, using the number you supply as the seed. This isn't random at all.
- A number greater than zero, such as Rnd(2), will return the next "random" number in the computer's sequence.
- Zero, or Rnd(0), will return the most recently generated random number again and again.

VBScript's random number generator uses a seed as its initial value, and then calculates pseudorandom numbers from there. Given the same seed, VBScript will always calculate the same sequence of random numbers every time.

Boolean Math

Boolean math is a special kind of logical math. If you know how to subnet TCP/IP addresses, you already know Boolean math, although you may not realize it. First, here are the basic Boolean operators that VBScript supports:

- NOT: Reverses a value from 0 to 1 (or False to True) or vice versa.
- AND: Returns a True if both values are True.
- OR: Returns a True if either value is True.
- XOR: Returns a True if one, but not both, values are True.

In VBScript, the value zero represents False; all other values represent True. Internally, VBScript generally uses -1 to represent True. To demonstrate Boolean math, try the following examples:

```
'Not

MsgBox NOT True

MsgBox NOT False


'And

MsgBox True AND False

MsgBox True AND True


'Or

MsgBox True OR False

MsgBox True OR True


'Xor

MsgBox True XOR False

MsgBox True XOR True
```

You should get the following results in message boxes:

- False (the opposite of True)
- True (the opposite of False)

Converting Numeric Data Types

As I mentioned earlier, VBScript happily converts data types for you when necessary. This process is called coercion, and it happens entirely behind the scenes as needed. There are times, however, when you want VBScript to handle data in a particular fashion. In those cases, you'll need to explicitly convert the data type.

For example, in Listing 7.2, you saw how I used the Rnd() function to generate a pseudorandom number. This number is a fraction, but I wanted a whole number, and so I used the Int() function to convert it to an integer. Other numeric conversion functions include

- Abs(): Returns the absolute value of a number, removing the positive or negative
- CBool(): Converts a value to either True or False
- CCur(): Converts a value to Currency
- CDbl(): Converts a value to a Double
- CSng(): Converts a value to a Single
- CInt() and Int(): Converts a value to an integer
- CLng(): Converts a value to a long integer

You'll often use these functions to convert user input to a specific data type. For example, if you have an input box that accepts the number of servers to shut down, you want to make sure that's a whole number, and not some fractional number, because a fraction wouldn't make sense. You might use something like this.

```
Dim vInput

vInput = InputBox("Shut down how many servers?")

If CInt(vInput) = vInput Then

 'Shut them down

Else

 MsgBox "You didn't type a whole number."

End If
```

In this case, I used CInt() to force vInput to be an integer, and then compared the result to the original value in vInput. If the two were the same, the original input was an integer and the script continues. If not, the script displays an error message and ends.

TIP

Never assume that some piece of data is a particular type. If the operation you are performing demands a specific

Converting Other Data Types to Numeric Data

You can also convert some non-numeric data into numeric data. For example, suppose you have the following in a script.

```
Dim vValue

vValue = InputBox("Enter a number of servers")
```

At this point, you've no idea what vValue contains. You can try to convert it to a number, though. Consider the following examples.

- If vValue contains "5 servers", CInt(vValue) would return 5, because the character 5 can be interpreted as an integer.
- If vValue contains "five", CInt(vValue) would return zero, because there are no numbers that can be converted to an integer.
- If vValue contains "5.2 servers", CInt(vValue) would return 5, because 5.2 can be interpreted as a number and the integer portion of that number is 5.

You can use any of the numeric conversion functions I've already covered to convert non-numeric data into numeric data. If vValue contains "five or 6 servers", CInt(vValue) would return zero, because the first characters cannot be interpreted as a number.

Review

VBScript's numeric and mathematical functions can be useful in a variety of situations. You can use basic math operators to perform simple math, and more advanced functions are available for complex geometric and algebraic operations. Boolean math plays a key role in logical comparisons, and VBScript provides a number of functions to convert numeric data into specific forms. You can also convert non-numeric data, such as strings, to numeric data in order to work with it.

COMING UP

In the next chapter, you'll build on your number manipulation skills to manipulate string data. Then, you'll move on to date and time data, Boolean data, and more. In fact, by the end of Chapter 9, you'll have mastered more than two-thirds of VBScript's functions!

Chapter 8. Manipulating Strings

IN THIS CHAPTER

It's a rare script that doesn't involve some sort of string data, such as user names, server names, and so forth. Working with string data is a key requirement. Fortunately, VBScript makes it easy, and almost fun, to perform even the most complex string operations. You'll learn to change strings, pull strings out of the middle of other strings, and much more.

Computer names, group names, user names, queries—strings are all around us in the world of administrative scripting. Learning to manipulate those strings is a key skill. You'll find yourself building file paths, working with server names, creating WMI queries, and much more. In fact, string manipulation is such a fundamental VBScript skill that you'll need to master it to some degree before you can start writing effective scripts.

Strings in VBScript

As you learned in Chapter 5, VBScript can store any type of data in a variable. String data is anything VBScript cannot interpret as another data type, such as a number or a date. Strings are simply any combination of letters, numbers, spaces, symbols, punctuation, and so forth. Often times, VBScript might interpret data as different types. For example, 5/7/2003 could be treated as a date or as a string, because it qualifies as both. In those instances, VBScript will coerce the data into one type or the other, depending on what you're trying to do with the data. Coercion is an important concept, especially when dealing with strings. For more information, refer to "What Are Variables?" in Chapter 5.

In your scripts, you'll always include strings within double quotation marks, which is how you let VBScript know to treat data as a string. For example, all of the following are acceptable ways to assign string data to a variable.

```
Var = "Hello"

Var = ""Hello""

Var = "Hello, there"

Var = vSomeOtherStringVariable
```

The second example is worth special attention. Notice that two sets of double quotes were used: This method will cause the variable Var to contain a seven-character string that begins and ends with quotes. Use this technique of doubling-up on quote marks whenever you need to assign the quote character itself as a part of the string.

VBScript refers to any portion of a string as a substring. Given the string Hello, one possible substring would be ell and another would be ello. The substring ello has its own substrings, including llo and ell. VBScript provides a number of functions for working with substrings. For example, you might write a script that accepts a computer name. The user might type just the name, such as Server1, or he might include a UNC-style name, such as \\Server1. Using VBScript's substring functions, you can get just the substring you want.

A large number of VBScript's intrinsic functions are devoted to string manipulation, and I'll cover most of them in this chapter. As a quick reference, here's each one, in alphabetical order, along with a quick description of what each does.

- Asc(). Returns the ASCII code for any single character.
- Chr(). Given an ASCII code, returns the corresponding character.
- CStr(). Converts a variable to a string.
- Escape(). Encodes a string for proper transmission as part of an Internet URL, so that strings such as "Hello world" become "Hello%20world."
- FormatCurrency(). Accepts a currency value and returns a properly formatted string. For example, formats 45.67 as $45.67.
- FormatDateTime(). Returns a properly formatted date or time string. For example, formats 4/5/2003 as April 5, 2003.

Working with Substrings

As I mentioned, string manipulation is often valuable when dealing with user input. For example, suppose you have a script that will work with a server, and you want the user to enter the server name in an input box. You might start with something like this.

```
Function GetServer()

 Dim sServer

 sServer = InputBox("Work with what server?")

 GetServer = sServer

End Function
```

NOTE

There doesn't seem much point in making this a special function at present, but bear with me. By the way, don't bother typing in these scriptlets yet—I'll be building on this example throughout the chapter.

The problem is that the user could type nearly anything. If this is a script that only you will be using, you can probably be sloppy and leave it as-is, knowing that you'll always type the right thing. However, if a junior administrator or technician will use the script, you should program some intelligence into it.

As an example, suppose the administrator typed a UNC-style name, such as \\Server1. If your script is expecting a simple name like Server1, the extra characters could cause problems. You can build your function to manipulate the string.

```
Function GetServer()

 Dim sServer

 sServer = InputBox("Work with what server?")


 'trim backslashes

 Do While Left(sServer,1) = "\"

  sServer = Right(sServer, Len(sServer) - 1)

 Loop


 'return result

 GetServer = sServer

End Function
```

In this new example, a Do…Loop construct is used to examine the leftmost character of sServer. As long as the leftmost character is a backslash, the loop will set sServer equal to sServer's rightmost characters. This is done with

Concatenating Strings

You've already learned about string concatenation, but let's look at it again. It's an important technique that you'll use repeatedly in your scripts.

For example, suppose you need to display a long, complicated message inside a message box. You could write a single MsgBox statement on a very long line of code, but that's harder to do and will make it tougher to maintain the script in the future. Instead, it's often easier to use string concatenation and line-continuation characters.

```
Dim sMessage

sMessage = "The server name you typed is invalid." & _

 vbCrLf & _

 vbCrLf & "Remember that all server names must : & _

 be seven characters " & _

 "long. The first three characters " & _

 must be the server's internal " & _

 "serial number. The second three characters " & _

 must be the three-" & _

 "character code for the office in which the " & _

 server is located. " & _

 "Finally, the last four characters indicate " & _

 the server's function:" & _

 vbCrLf & vbCrLf & "FILES = File Server" & _

 vbCrLf & vbCrLf & _

 "DOMC = Domain Controller" & vbCrLf & vbCrLf & _

 "SQLS = SQL Server" & vbCrLf & vbCrLf & _

 "Please try again."

MsgBox sMsg
```

I can't even show you the alternative in this book—there's no way for me to spread a single line of code across multiple pages!

String concatenation is also useful when you're working with variables. For example, suppose you need to generate some kind of unique password for new users. The following function might be used in a script that creates new user accounts.

```
Function MakePassword(sUserName)

 Dim sPassword

 sPassword = Left(sUserName,1)
```

Changing Strings

VBScript includes a wide array of functions designed to change strings. I'll start with LCase() and UCase(), which change a variable to lower- or uppercase letters, respectively. Try running the following scriptlet to see how these functions work.

```
Dim sInput

sInput = InputBox("Enter a string to try.")


MsgBox "All upper: " & UCase(sInput)

MsgBox "All lower: " & LCase(sInput)
```

These functions can be very useful when dealing with case-sensitive strings, such as passwords, WMI queries, and so forth. Using these functions, you can ensure that the case of the strings is exactly what you need for whatever you're doing.

Combining these functions with the substring functions lets you perform some very powerful tricks. For example, the following function will accept a full user name, such as "john doe," and convert it to the proper name case, where the first letters of each name are capitalized, no matter how you capitalize the input.

```
Dim sUserName


'get the user name

sUserName = InputBox("Enter user name")


'does it contain a space?

If InStr(1, sUserName, " ") = 0 Then


 'no - error message!

 MsgBox "Name must contain a space."


Else


 'display the name case version

 MsgBox "Proper case is " & NameCase(sUserName)


End If
```

Formatting Strings

VBScript provides several functions designed to format strings—and other data types—into specially formatted strings. For example, suppose you have a function that calculates the total up time for a server, and you want to display that information as a percentage. The following script is an example of how VBScript lets you format the output.

```
Dim iUpHours, iDownHours

iUpHours = InputBox("How many hours has the server " & _

 been up?" & _

 " Fractional numbers are OK.")

iDownHours = InputBox("How many hours has the server " & _

 been down?" & _

 " Fractional numbers are OK.")


Dim sResult

sResult = CalcDownPerc(iUpHours, iDownHours)

MsgBox "The server has been down for " & _

 sResult & " of the " & _

 "time it has been up."


Function CalcDownPerc(iUpHours, iDownHours)

 Dim iPerc

 iPerc = iDownHours / iUpHours


 Dim sDisplay

 sDisplay = FormatPercent(iPerc, 4)


 CalcDownPerc = sDisplay

End Function
```

In this example, FormatPercent() is used to format the contents of variable iPerc so that the result has four digits after the decimal place, and the result may have a leading zero before the decimal depending upon the computer's locale settings.

Another popular formatting function is FormatDateTime(). In the next example, suppose that variable dLastLogon contains a user's last logon date.

Converting Other Data Types to String Data

First, keep in mind that the formatting functions I introduced you to in the previous section will return a string value. So, if you use something like this:

```
Dim iNumber, sString

iNumber = 5

sString = FormatPercent(iNumber, 2)

MsgBox sString
```

variable sString will contain a string value, because that's what FormatPercent() returns. Technically, the formatting functions are a sort of specialized string conversion function, too.

```
Dim dDate, sString
```

VBScript does provide a general string conversion function: CStr(). This function simply takes any type of data—numeric, date/time, currency, or whatever—and converts it to a string. The function works by taking each character of the input data and appending it to an output string. So the number 5 will become "5," the number -2 will become "-2," and so forth. Dates and times are converted to their short display format. For example, try running this.

```
dDate = Date()

sString = CStr(dDate)

MsgBox sString
```

The result should be a short formatted date, such as "5/26/2003."

NOTE

If your computer is displaying short dates with a two-digit year, you probably have an outdated version of the Windows Script Host or an incredibly old operating system. All newer versions of Windows and the Windows Script Host display four-digit years to help eliminate future recurrences of the infamous "Y2K bug."

Review

Believe it or not, you've probably covered half of VBScript's functions in this chapter. That alone should help you realize how important string manipulation is, and may explain the spinning feeling in your head right now! Don't worry—string manipulation, like everything else involved in scripting, becomes easier with practice.

For now, keep in mind the basic functions for working with substrings, such as Right(), Left(), Mid(), and InStr(). String concatenation using the & operator is also important, as is the ability to change strings with functions like Replace(). Finally, string conversion functions—especially CStr()—can help make your scripts less error-prone, while enabling you to work with a broad variety of data.

Your string manipulation skills will serve you well in other areas of VBScript, such as date and time manipulation, Active Directory querying, Windows Management Instrumentation, and more.

COMING UP

In the next chapter, I'll explore other types of data, such as dates and times, bytes, and a couple of others, that you might find in your scripts. Then, in Chapter 10, I'll look at adding logic to your scripts with control-of-flow statements.

Chapter 9. Manipulating Other Types of Data

IN THIS CHAPTER

Dates and times, arrays, bytes, and more—all data types you may run across from time to time, especially when you're writing more complex scripts that involve Windows Management Instrumentation. I'll explain how each of these data types works, and how you can work with them in VBScript.

In the prior two chapters, you learned a lot of about string and numeric data. In this chapter, I'll cover everything else—the lesser-used data types that are nonetheless so important to VBScript. You'll find yourself using these data types most frequently in complex scripts. For example, I'll begin with date and time data, which you'll use frequently in many Windows Management Instrumentation (WMI) scripts. I'll also cover byte data, which is a lot less common in administrative scripts, but worth knowing about in case you need it. Finally, I'll cover arrays, which aren't really a data type at all. They're a special type of variable capable of holding multiple values, and you'll use them in many of the scripts you write.

Working with Dates and Times

Dates and times allow your scripts to interact more precisely with the real world. You can copy or move files based on their "last changed" date, delete users based on the last time they logged on, and so forth. Next to strings and numbers, dates and times are the third most common data type that you'll use in your scripts.

Dates and Times in VBScript

VBScript stores dates and times in a serial number format that looks like a large decimal number. The serial number counts the number of milliseconds that have elapsed since January 1, 100 C.E., and can represent dates and times up to December 31, 9999. The integer portion of a date serial number—the portion before the decimal point—is used to represent days (and thus, months and years), whereas the fractional portion—the part after the decimal point—represents milliseconds (and seconds, minutes, and hours).

VBScript includes a number of functions for working with dates and times. For example, the DatePart() function analyzes a date and returns just the specified part of it. DatePart("yyyy", Date()), for example, returns the year portion of the current date. DatePart() accepts a number of different strings, which tell it which portion of the date you're interested in.

- "yyyy" returns the year.
- "q" returns the quarter of the year.
- "m" returns the month.
- "y" returns the Julian date, which is the number of days that have elapsed since the beginning of the year.
- "d" returns the day as a number.
- "w" returns the weekday, such as "Monday."
- "ww" returns the week of the year.
- "h" returns the hour.
- "n" returns the minute. Don't confuse this with "m," which actually returns the month.
- "s" returns the second.

The second parameter of DatePart() can be anything VBScript recognizes as a date or time, including string variables that contain date or time information, such as "1/1/2004" or "10:26 P.M."

Getting the Date or Time

VBScript has a number of functions that return the current date or time, or portions thereof.

Working with Arrays

An array is a collection of values assigned to a single variable. Normal variables can hold just one value. For example:

```
Dim sMonths

sMonths = "January"
```

In this example, sMonths could be changed to contain "February," but doing so would eliminate "January" from the variable's contents. With an array, however, a single variable can contain multiple values. For example:

```
Dim sMonths(12)

sMonths(1) = "January"

sMonths(2) = "February"

sMonths(3) = "March"

sMonths(4) = "April"

sMonths(5) = "May"

sMonths(6) = "June"

sMonths(7) = "July"

sMonths(8) = "August"

sMonths(9) = "September"

sMonths(10) = "October"

sMonths(11) = "November"

sMonths(12) = "December"
```

This capability to assign multiple values to a single variable can come in handy in a number of scripting situations.

Arrays in VBScript

VBScript supports multidimensional arrays. For example, suppose you declare a variable using Dim sData(5,4). This creates a two-dimensional variable. The first dimension can hold six data elements, whereas the second dimension can hold five. Note that elements always begin numbering at zero. I sometimes find it easier to imagine a two-dimensional array as a table of elements. The columns represent one dimension, whereas the rows represent another dimension.

sData	0	1	2	3	4
0	Harold	Todd	Lura	Ben	Mary
1	Cyndi	David	Deb	Amy	Barb

Working with Bytes

A byte variable can contain a single byte of data—that is, a number from 0 to 255. Doesn't sound very useful, does it? Bytes aren't often used alone, though; they're often used in arrays, where a single byte array can represent a stream of binary data. For example, files on a computer's hard drive are a simple one-dimensional array of bytes. A file that's 1KB in length has 1,024 elements in its array, and can be contained with a byte array in an administrative script.

Bytes in VBScript

Your most frequent use for byte variables will be to pass data to WMI functions that require a byte array. You'll usually work with bytes in the form of an array, where the data inside the array represents a file or some other binary data. Still, bytes are reasonably rare in administrative scripts, which is why I won't bore you with a long example. You'll see one or two examples elsewhere in this book that use bytes; I'll call your attention to them and explain them in a bit more detail at that time.

Converting Byte Data

The CByte() function converts data to a byte. Generally, only numeric data in the range of 0 to 255 can be successfully converted to a byte.

```
Dim iDouble, bByte

iDouble = 104.76

bByte = CByte(iDouble)
```

In this example, bByte now contains the value 105, which is the closest whole number to what iDouble contains.

Review

Dates, times, bytes, and arrays are used less often, but are important types of data in VBScript. Although you may not have an immediate need for them in your administrative scripts, keep them in mind. When you do run into them in the future, or when you see them in the example scripts I'll present throughout this book, you can refer back to this chapter to learn more about them or to refresh your memory.

Bytes, dates, and times use conversion and manipulation functions very similar to those you've learned to use with string and numeric data. Date and time data can also be used with the unique calculation functions DateAdd() and DateDiff(). Arrays, however, aren't really a data type at all; they're a way to collect multiple values into a single variable. Arrays can be strings, numbers, dates, times, or bytes. You can create and manipulate arrays with functions like Join(), Split(), and ReDim.

COMING UP

Now it's time to add some intelligence to your scripts, teaching them how to react to changing conditions and to change their behavior based on your input. In the next chapter, I'll introduce you to control-of-flow statements and constructs, and, you'll be nearly finished with your VBScript tutorial. After a quick overview of the built-in scripting objects and the file system objects, I'll show you how to pull everything together in your first script.

Chapter 10. Controlling the Flow of Execution

IN THIS CHAPTER

You've learned almost everything you need to know to write great scripts. The last step in your VBScript crash course is to add complexity and logic to your scripts by using control-of-flow constructs.

At this point, you should know enough VBScript to write some useful administrative scripts. In fact, the previous few chapters contained some great example scripts that you should be able to put right to use, in addition to using them as reference examples.

What you lack at this point, and what I'll cover in this chapter, is a way to make your scripts automatically respond to certain conditions, and execute different lines of script accordingly. For example, suppose you need to write a script that tells you which user has a particular file open on a file server. Your script must be able to iterate through all of the open resources on a server to find the one you're interested in, and then iterate through the list of users that have the resource open, displaying that information to you. Such a script would require certain lines of code to be repeated over and over, while requiring other lines of code to be executed only if certain conditions are true (such as if the current server resource is the one you're interested in).

VBScript includes control-of-flow statements that give your scripts the necessary logical-evaluation capabilities. In this chapter, you'll learn how they work, and see some examples of how to use them in your scripts.

Conditional Execution

Many administrative scripts that you write will execute some simple, straightforward task that doesn't require any decisions. Other scripts, however, will be more complex, and will require your scripts to evaluate conditions and values and make a decision about what to do. VBScript conditional execution statements make this possible, giving your scripts a form of intelligence and decision-making capabilities.

If…Then

The most common conditional execution statement is the If…Then construct. It's referred to as a construct because it involves more than a single statement or more than even a single line of code. Here's a very simple example.

```
Dim iMyNumber

iMyNumber = InputBox("Enter a number from 1-100")



If iMyNumber < 1 Or iMyNumber > 100 Then

 MsgBox "I said 1 to 100!"

Else

 MsgBox "Thank you!"

End If
```

The script declares a variable named iMyNumber, and then uses InputBox() to retrieve user input. Next, the script uses an If…Then construct to evaluate the input. Here's how it works.

- First, VBScript evaluates the two logical expressions in the If statement. Does iMyNumber contain a number that is less than one? Does it contain a number that is more than 100? If either of these two conditions are true, VBScript will execute the code following the Then statement. VBScript will accept either of these two conditions because they're connected with an Or statement, which means either one of them being true is acceptable.
- If neither of the If conditions are true, VBScript looks for an alternate execution path, which it finds after the Else statement. VBScript executes that code instead of the code following Then.
- Conditional execution stops whenever another portion of the If…Then construct is reached.

Boolean Operators

And and Or are examples of Boolean operators. These operators are similar to mathematical operators, except that instead of resolving a value, these resolve a logical condition and return a True or False value.

For example, suppose you have a variable named iNumber, which contains the value 4. The logical condition iNumber > 1 And iNumber < 100 would evaluate to True, because both subconditions evaluate to True. Similarly, the logical condition iNumber > 1 Or iNumber < 0 would also evaluate to True, because one of the two subconditions evaluates to True.

Loops

There will be times when you want VBScript to repeat the same task over and over. Perhaps you're having it evaluate a number of different files, or perhaps you simply want to make the computer beep a lot and annoy the person in the cube next to yours! Regardless of your motives, VBScript provides statements that make repetitive execution easy, and gives you full control over how many repetitions VBScript performs.

Do While…Loop and Do…Loop While

The Do While…Loop construct is used to execute a given section of code so long as a specified logical condition is true. Here's one way in which Do While…Loop can be used.

```
Dim iNumber

Do

 iNumber = InputBox("Please enter a number.")

Loop While Not IsNumeric(iNumber)

MsgBox "Thank you!"
```

This short script is an excellent example of collecting and validating user input. It starts by declaring a variable, iNumber. Next, VBScript enters the Do loop. Notice that there are no logical conditions specified with Do; it's on a line by itself, meaning VBScript will always execute the code within the loop.

Within the loop, VBScript uses an input box to collect user input, and assigns that input to the variable iNumber. The Loop statement contains the logic of the Do While…Loop construct: Not IsNumeric(iNumber). IsNumeric() is a function that evaluates a variable and returns True if the contents are numeric, and False otherwise. The Not Boolean operator tells VBScript to reverse the output of IsNumeric. So, if iNumber contains a number, the result of Not IsNumeric(iNumber) will be False, the opposite of what IsNumeric(iNumber) would return.

The Loop While statement tells VBScript to return to the Do statement whenever the logical expression is True. In this case, the logical expression will be True only if iNumber doesn't contain a numeric value. In other words, VBScript will continue asking for input repeatedly until that input is numeric.

When the input is finally numeric, VBScript stops executing the loop and responds with a message box reading, "Thank you!" and the script ends.

When you include a logical expression with Loop, VBScript always executes the code within the loop at least once. That's because VBScript executes code in the order in which it finds it, so it doesn't get to the Loop until it has already executed the code within the loop at least once. There may, however, be times when you don't want the script in the loop executed at all, unless a certain condition is true to begin with. For example, suppose you've written a script that opens a text file of unknown length. The file itself is represented by an object name oFile, and that object has an EndOfFile property that will be True when the end of the file is reached. You can use the Read method of the oFile object to read data from the file. In that case, you might use a section of script like this one to read through the entire file.

```
' assumes oFile is some kind of file object

' that is opened for reading

Dim sData

Do While Not oFile.EndOfFile
```

Putting It All Together

With all of these loops and conditional execution constructs under your belt, you're probably ready to see them in action!

➤➤ Who Has a File?

Listing 10.3 is a sample script that shows you which user or users has a particular file open on a file server.

Listing 10.3. WhoHasFile.vbs**. Shows who has a particular file open.**

```
' first, get the server name we want to work with

varServer = InputBox ("Server name to check")


' get the local path of the file to check

varFile= 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 objFS = GetObject("WinNT://" & varServer & _ "/lanmanserver,fileservice")


' scan through the open resources until we

' locate the file we want

varFoundNone = True


' use a FOR...EACH loop to walk through the

' open resources

For Each objRes in objFS.Resources


     ' does this resource match the one we're looking for?

     If objRes.Path = varFile then

          ' we found the file - show who's got it

           varFoundNone = False

          WScript.Echo objRes.Path & " is opened by " & objRes.User

     End If

Next
```

Review

In this chapter, you've learned to write scripts that can evaluate various criteria and change the execution of the script accordingly. You can use the If…Then construct to evaluate logical conditions and execute different sections of script depending on those conditions. Select…Case is a sort of super If…Then construct, allowing your script to evaluate a number of possible conditions and execute script code accordingly.

You also learned how to write loops, such as Do…Loop and For…Next. These constructs allow your script to execute specific lines of code over and over, while evaluating logical criteria to determine when the repetitive execution should stop. Finally, you learned how to use For Each…Next to iterate through a collection of objects, making it easier to work with collections.

That's about all there is to VBScript! You've already learned about functions, statements, objects, and variables (in Chapter 5), which provide the basis of VBScript's operations. You also learned how to collect user input and display messages (in Chapter 6), which provides your script with interactivity. Chapters 7, 8, and 9 covered how to manipulate various types of data within your script. With all of that under your belt, you're ready to start "gluing together" various operating system objects and writing truly functional administrative scripts.

COMING UP

Incredibly, you have finished learning VBScript. Now, you can start learning about the various objects that provide access to key operating system features. You'll begin with the built-in scripting objects in the next chapter, and move on to the FileSystemObject in Chapter 12.

Chapter 11. Built-in Scripting Objects

IN THIS CHAPTER

You've now seen most of the VBScript language, so it's time to start working with those "objects" that you've heard so much about. I'll begin with the objects that are built into the scripting engine itself, and you'll find that they're quite useful in a number of situations.

I've already described how VBScript's real value is as a sort of electronic "glue," which you can use to piece together the many objects of the Windows operating system. Windows Management Instrumentation (WMI) and Active Directory Services Interface (ADSI) are good examples of operating system functionality that you can access by using VBScript. The Windows Script Host (WSH) even has its own built-in object library, and these objects allow you to perform some powerful tasks.

In this chapter, you'll learn to use the WSH Network object, which provides access to the computer's network environment; the Shell object, which allows you to manipulate Explorer and other shell-related information; and the Shortcut object, which allows you to work with Explorer shortcuts and Internet links.

All of these objects can be used in a wide variety of situations, but I think you'll find them more useful in logon scripts. The Network object, for example, allows you to map network drives and printers, which is perhaps the most common job of a logon script.

Chapter 29 contains additional logon script examples for both NT and Active Directory domains, and includes some suggestions for using logoff scripts.

The WScript Object

All of these objects are accessed through the top-level WScript object. You've already seen WScript in use in Chapter 6, where I showed you how WScript.Echo can be used to produce both command-line output and message boxes, depending on whether you are using Cscript.exe or WScript.exe to execute your script. The WScript object is the only one your scripts get free, meaning you don't have to explicitly create a reference to it. WScript is always available when you're running a script in WSH.

In addition to Echo, the WScript object has new methods and properties that may be useful to you in your scripts. For example, you can execute the WScript.Sleep method, passing a specific number of milliseconds, to have your script pause its execution.

```
'Pause for 5 seconds

WScript.Sleep 300000
```

You can have your scripts immediately stop execution and exit, if you want.

```
If varInput = "" Then

 WScript.Quit

End If
```

In this example, the script will immediately exit if variable varInput is empty. You can also ensure that your scripts have a timeout. By default, WSH will continue executing your scripts forever; you may, however, want to automatically have your scripts end if they don't complete within, say, 30 seconds. That way, a script that has the chance of entering some endless loop, or trying to connect to a remote computer that isn't available, will eventually stop running. To do so, simply set a timeout value.

```
'Specify a timeout in seconds

WScript.Timeout = 30
```

Most importantly, the WScript top-level object provides access to important child objects that you'll need to use in many of your scripts.

The Network Object

The WScript.Network object provides access to drive and printer mapping functions, as well as access to network information, such as the current user and domain names. You must explicitly create an instance of the Network object in order to use it.

```
'Create reference

Dim oNetwork

Set oNetwork = CreateObject("WScript.Network")
```

When created, you can use the object in your scripts.

Overview

The Network object is designed primarily for use in logon scripts, where you'll need to map both drives and printers. Obviously, it has uses elsewhere, but logon scripts demonstrate its usefulness. The Network object provides three functions.

1. Working with network drives, including mapping and unmapping them, as well as enumerating them.
2. Working with network printers, including mapping and unmapping them, as well as enumerating them.
3. Providing access to the network environment's information, such as the current user and domain names.

NOTE

All of the examples in this section assume that you've created a variable named oNetwork and set it to be a reference to the WScript.Network object.

By the way, if you're in a rush to get to WMI, you should know that it's not the be-all and end-all of scripting. In fact, most of the functionality offered by the Network object, particularly mapping network drives, isn't possible through WMI.

Methods and Properties

The MapNetworkDrive object has several different methods for working with drives and printers, and three properties for obtaining network environment information.

MapNetworkDrive

You'll most often see drives mapped using a simplified version of the MapNetworkDrive method.

```
'map a drive

oNetwork.MapNetworkDrive "Z:", "\\Server1\public"
```

The Shell Object

The Shell object must be explicitly created and assigned to a variable, just like the Network object. In this section, I'll assume that your scripts already contain the following code.

```
'Create shell object

Set oShell = CreateObject("WScript.Shell")
```

Overview

You can use the Shell object to execute external applications, work with special folders and shortcuts, manipulate environment variables, write to the event log, read and write to the registry, create timed dialog boxes, and even send keystrokes to another application. Shell is sort of the catchall of the WSH, containing a number of useful functions.

Methods and Properties

The Shell object's methods and properties provide access to its functionality. Many of these methods and properties are complementary, so I'll discuss them together in the following sections.

Run and Exec

Scripting can't do it all. That's an important thing to remember. I always set myself a research time limit: If I can't figure out how to do something in script within 30 minutes of searching on the Web, I'll do it whatever way I already know how. If that means launching an external command line, so be it. A good example is setting NTFS permissions on files and folders. You can absolutely do that from within WMI, but it's a thankless, complicated task. I've taken the pain to figure it out a few times, but it's almost always easier to just launch Cacls.exe with the appropriate parameters, so that's what I usually do, using Run and Exec.

Both methods launch new applications in separate processes. With Run, that process is completely detached from your script, and your script will have no access to it. Most of the time, that's fine. With Exec, your script has access to the new process' input and output streams, meaning you can read the output of command-line utilities or other applications into your script, and then do something else based on what happened.

Here's how you can use Run to launch the DIR command.

```
Call oShell.Run("cmd /c dir " & _

 "/a")
```

Notice that you have to launch the command-line processor, CMD, first; you can tell it to run DIR for you. This is an interesting technique, but not useful, as your script has no way to get at the DIR results. You could have DIR redirect its output to a text file, and then read in the text file…but what a pain. There's an easier way.

```
Dim oExecObject, sDir

Set oExecObject = oExec("cmd /c dir /a")

Do While Not oExecObject.StdOut.AtEndOfStream

 sDir = sDir & oExecObject.StdOut.Readline()
```

The Shortcut Object

Shortcut objects are created by using the Shell object's CreateShortcut method. This method only specifies the final location for the shortcut; it doesn't allow you to specify the shortcut's own properties. To do that, you modify the properties of the Shortcut object, and then call the Shortcut object's Save method to save your changes.

Methods and Properties

The Shortcut object offers the following properties.

- Arguments. These are any command-line arguments that should be passed when the shortcut is launched.
- Description. A description of the shortcut.
- FullName. This is a read-only property and returns the full name of the target application.
- HotKey. The hot key that can be used to launch the shortcut from the keyboard. You can use any letter, number, or function key (F1 to F12). You can also specify Control, Alt, or Shift keys, such as Alt+F9.
- IconLocation. The name of an icon file, along with an index to a specific icon, that should be used for the shortcut.
- TargetPath. The complete path and filename to the target application. UNCs are acceptable.
- WindowStyle. Specifies the starting window style for the shortcut when launched.
- WorkingDirectory. Sets the working directory for the application launched by the shortcut.

You can create two types of shortcuts:

1. Standard shortcuts have an LNK filename extension and generally point to applications on the local computer or network.
2. Internet shortcuts have a URL filename extension and point to Web sites.

You'll see examples of both in Listing 11.2.

Practical Application

Listing 11.2 shows an example script that creates both a normal application shortcut and a URL shortcut.

Listing 11.2. Shortcuts.vbs**. Creates shortcuts on the user's desktop.**

```
' this sample creates two shortcuts on the current user's desktop
```

Review

In this chapter, you've seen how the built-in WScript, Network, Shell, and Shortcut objects work. With these, you'll be able to write effective logon scripts, utility scripts, and much more. Perhaps more importantly, you've seen examples of how VBScript can be used to call on objects that are provided by the Windows operating system. Throughout the rest of this book, you'll be building on that skill to utilize more complex and powerful objects, including ADSI and WMI, to accomplish even the most difficult administrative tasks.

COMING UP

I'll continue working with objects in the next chapter by introducing you to the FileSystemObject. Then, in Chapter 13, I'll show you how to put together everything you've learned so far: You'll design, write, test, and debug an entire script, all from scratch.

Chapter 12. Working with the File System

IN THIS CHAPTER

Manipulating files and folders is one of the most common things an administrative script needs to do. Windows provides the script-friendly FileSystemObject to make it easy for administrative scripts to access the file system on your computer.

You'd be surprised how often you might need to access a computer's file system from within an administrative script. For example, a script that adds new users to the domain might need to read those names from a script, or might need to write out new passwords into a file. A script designed to query TCP/IP addresses from workstation computers will need to write that information somewhere—why not a text file? File system access is almost a prerequisite for any number of useful scripts, even ones that don't have a basic goal of manipulating files or folders. Fortunately, the Windows scripting library includes the FileSystemObject, or FSO, which provides easy access to the drives, files, and folders on your computer.

The FileSystemObject Library

The FSO is actually an object library, which simply means that it's made up of bunches of other objects. These other objects represent things like files and folders on your computer. As with any other object—or library—you start working with the FSO in a script by declaring a variable and creating an instance of the object.

```
Dim oFSO

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

TIP

Where do I get these object names? Generally, from their documentation. In the case of the FSO, the MSDN Library contains complete documentation under its Platform SDK section. If you're using the Library, either from CD, DVD, or http://msdn.Microsoft.com/library, look under Platform SDK first. Then look under Tools and Scripting, expanding each section as you go. Alternatively, open the index and simply type FileSystemObject to jump straight to an overview.

One look at the FSO's documentation and you may wonder what you've gotten yourself into. The FSO contains an almost bewildering number of properties, objects, and methods for you to work with. Don't let this bounty of options overwhelm you! The FSO only has four basic objects that you'll work with.

- A Drive object represents a drive on your system. Drives can include removable drives, fixed drives, mapped network drives, and so forth.
- A Folder object represents a folder in the file system.
- A File object represents—you guessed it—a file.
- A TextStream object represents a stream of text, which is a fancy way of describing a text file. More precisely, a TextStream allows you to pull (or stream) text in and out of a file, providing a handy way to work with the contents of text files.

All of the FSO's other methods, properties, and objects are designed for working with these four basic objects. I'll cover each of these objects in their own section, along with their associated properties and methods.

TIP

One of the things you often have to worry about with objects is whether the objects will be available on every machine that you want to run your script on. With the FSO, that's not a problem: It's implemented in Scrrun.dll, the Scripting Runtime, which is present on all Windows 2000 and later computers, Windows Me, and generally on Windows 98. In fact, on Windows 2000 and later, the file is under Windows File Protection and cannot easily be removed.

Working with Drives

Drive objects represent the logical drives attached to your system, including network drives, CD-ROM drives, and so forth. Drives also provide an entry point into each drive's file system, starting with the root folder of the file system hierarchy. Because the Drive object represents one of the simplest aspects of the file system, it's one of the simplest objects in the FSO.

The method you'll use most with drives is GetDrive, which returns a Drive object given a specific drive letter. For example, to obtain a Drive object that represents your C: drive:

```
Dim oDriveC, oFSO

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

Set oDriveC = oFSO.GetDrive("C:")
```

You can also use the FSO's root-level Drives collection to iterate through all of the drives attached to your system.

```
Dim oFSO, oDrive

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

For Each oDrive In oFSO.Drives

 MsgBox "Drive " & oDrive.DriveLetter & _

  " has a capacity of " & oDrive.TotalSize & " bytes " & _

  " and is drive type " & oDrive.DriveType

Next
```

Working with Drive Objects

The previous example illustrates the use of some of the Drive object's properties. The full list includes the following.

- AvailableSpace and FreeSpace return the number of bytes available on the disk. FreeSpace returns the amount of free space on the drive; AvailableSpace returns the amount available to the user running the script. File quotas and other concerns can result in a difference between these two properties.
- DriveLetter returns the drive's logical letter. Note that not all drives must have a drive letter, especially in Windows 2000 or later, although most of the time they will.
- DriveType tells you what kind of drive you're looking at. This property returns a number corresponding to a specific drive type.
- FileSystem tells you what kind of file system the drive uses. This is a string, such as FAT, NTFS, or CDFS (used for optical media like CDs and DVDs).
-

Working with Folders

Folders offer up a bit more complexity. First, the FSO itself offers more methods for manipulating specific folders.

- CopyFolder copies a folder.
- CreateFolder creates a new folder.
- DeleteFolder removes a folder permanently. Note that the deleted folder doesn't ever make it to the Recycle Bin, and there's no "Are you sure?" prompt.
- FolderExists, like DriveExists, returns a True or False indicating whether the specified folder exists.
- GetFolder accepts a complete folder path and, if the folder exists, returns a Folder object that represents the folder.
- GetParentFolderName accepts a complete folder path and returns the name of its parent folder. For example, GetParentFolderName("C:\Windows\System32") would return "C:\Windows".
- GetSpecialFolder returns the complete path to special operating system folders. For example, GetSpecialFolder(0) returns the path for the Windows folder. Use 1 for the System32 folder, and use 2 for the system's temporary files folder.
- MoveFolder moves a file a folder.

The following example illustrates a few of these base functions.

```
Dim oFSO

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



Dim oFolder

If oFSO.FolderExists("C:\MyFolder") Then

 Set oFolder = oFSO.GetFolder("C:\MyFolder")

Else

 oFSO.CreateFolder "C:\MyFolder")

 Set oFolder = oFSO.GetFolder(C:\MyFolder")

End If

MsgBox oFSO.GetParentFolderName(oFolder.Path)
```

Working with Files

Files, of course, are the most granular object you can work with inside the FSO, and they're relatively uncomplicated. As with Drive and Folder objects, the FSO itself has some useful methods for working with files:

- CopyFile
- DeleteFile
- FileExists
- GetFile
- MoveFile

These all work similarly to their Folder object counterparts, allowing you to obtain a reference to a file (GetFile), check for a file's existence (FileExists), and copy, delete, and move files. You can also create files, which is a process I'll cover a bit later in this chapter.

Working with File Objects

File objects themselves have a few methods.

- Copy copies a file.
- Delete removes a file without warning and without using the Recycle Bin.
- Move moves a file.
- OpenAsTextStream opens a file for reading (which I'll cover in the next section).

Properties of the File object include

- Attributes
- DateCreated
- DateLastAccessed
- DateLastModified
- Drive

Reading and Writing Text Files

The FSO provides basic functionality for reading from, and writing to, text files. If you think of a text file as one long string of characters, you'll have an idea of how the FSO views text files. In fact, that long string of characters is what the FSO calls a TextStream. TextStream objects are how you get text into and out of text files.

The FSO has two basic methods for creating a TextStream: CreateTextFile and OpenTextFile. Both methods require you to provide a filename, and allow you to specify optional parameters, such as whether to overwrite any existing file when creating a new one. Here's an example.

```
Dim oFSO, oTS

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

Set oTS = oFSO.CreateTextFile("c:\test.txt")

oTS.WriteLine "Hello, World!"

MsgBox "All Done!"

oTS.Close
```

As you can see, the result of the CreateTextFile method is a TextStream, which is assigned via the Set command to variable oTS. TextStream objects have some properties and methods all their own. First, the methods:

- Write writes one or more characters to the file.
- WriteLine writes one or more characters and follows them with a carriage return/linefeed combination, thus ending the line as you would in Notepad when you press Enter.
- Close closes the TextStream.
- Read reads a specified number of characters from a TextStream.
- ReadLine reads an entire line of characters—up to a carriage return/linefeed.
- ReadAll reads the entire TextStream.

One useful property of a TextStream is AtEndOfStream, which is set to True when you've read all the way through a text file and reached its end.

Files must be opened either for reading, writing, or appending. When a file is opened for reading, you can only use the Read, ReadLine, and ReadAll methods; similarly, when the file is opened for writing or appending, you can only use Write or WriteLine. Of course, you can always use Close.

NOTE

Appending a file simply opens it and begins writing to the end of the file, while leaving the previous contents intact. This can be useful for writing messages to an ongoing log file.

Other FSO Methods and Properties

The base FSO object offers a few other useful methods and properties that you may need from time to time.

The first is the BuildPath function. It accepts components of a file or folder path and appends them together. Normally, you could do that with the simple & concatenation operator, but BuildPath actually worries about getting backslashes in the right place. So, consider this example:

```
Dim sFolder, sFile

sFolder = "C:\Windows"

sFile = "MyFile.exe"


Dim oFSO

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

MsgBox sFolder & sFile

MsgBox oFSO.BuildPath(sFolder,sFile)
```

The first message box displays "C:\WindowsMyFile.exe", which isn't right—it is missing the backslash in the middle. The second message box, which uses BuildPath, displays the correct "C:\Windows\MyFile.exe" because the BuildPath function figured out that a backslash was necessary.

While working with paths, you may also have a need to get the absolute or base path name, and the FSO's GetAbsolutePathName and GetBaseName methods will do it for you. Here's an example.

```
Dim oFSO, sPath1, sPath2

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

sPath1 = "C:\Windows\System32\Scrrun.dll"

sPath2 = "..\My Documents\Files"


MsgBox oFSO.GetAbsolutePathName(sPath1)

MsgBox oFSO.GetAbsolutePathName(sPath2)


MsgBox oFSO.GetBaseName(sPath1)

MsgBox oFSO.GetBaseName(sPath2)
```

The result of this is four message boxes.

- "C:\Windows\System32\Scrrun.dll": There's no difference between the input and output, because the input in this case is already a complete, unambiguous path.

Creating a Log File Scanner

➤➤ The Log File Scanner

Listing 12.6 shows the complete log file scanner.

Listing 12.6. ScanLog.vbs. **Scans for "500" errors in an IIS log file.**

```
' Scan a log file from a webserver for
' occurrences of " - 500" which indicates an
' internal server error
' get the log file
Dim varLogFile
varLogFile = InputBox ("Enter the complete " & _
 "path and filename " & _
 "of log file to scan.")

' create filesystemobject
Dim oFSO
Set oFSO = WScript.CreateObject("Scripting.FileSystemObject")

' open file into a TextStream object
Dim oTS
Set oTS = oFSO.OpenTextFile (varLogFile)

Dim oTSOut
Set oTSOut = oFSO.CreateTextFile ("c:\errors.htm")



' begin reading each line in the textstream
dim varLine, varFoundNone
varFoundNone = true
Do Until oTS.AtEndOfStream
 varLine = oTS.ReadLine

 ' contains a 500 error?
```

Review

In this chapter, you learned about the scripting FileSystemObject, which can be used to manipulate the files and folders on a computer. You learned about the object's flexible object hierarchy, which emulates the hierarchy of files and folders on your computer. You saw an example script of how the FileSystemObject can be used to move and copy files, delete them, and even open and read through existing text files. The FileSystemObject is flexible enough to earn a place in many of your scripts, and you'll see it in many of the example scripts in upcoming chapters.

COMING UP

You have finished learning VBScript, and you're ready to pull together everything you have learned up to now. In the next chapter, you'll design, build, and test a script designed to automate the process of archiving and rotating IIS log files. In Chapter 14, you'll begin working with advanced scripting topics, including Active Directory.

Chapter 13. Putting It All Together: Your First Script

IN THIS CHAPTER

You've learned how to use VBScript and several operating system objects; in this chapter, you'll bring together everything you know to design, write, and test an IIS log file rotation tool.

You've already learned just about all the VBScript commands, statements, and functions that you'll need to write administrative scripts. You've learned about some of the built-in scripting objects, and you've had a chance to work with the Windows FileSystemObject. Altogether, that's plenty of information and experience to start writing useful administrative scripts!

In this chapter, you'll design and write a tool that rotates IIS log files. As you probably know, IIS can create a log file for each Web site it operates, and by default, it starts a new log file each day. Your rotation tool will copy the previous day's completed log file to an archival folder for long-term storage. At the same time, the script will delete the oldest log file, keeping a rolling thirty days' worth of log files in the archival folder.

NOTE

To keep things interesting, I'm going to introduce a couple of logic errors into the scripts in this chapter. These scripts should run more or less without error, but they'll have unexpected results due to the way they're written. If you spot the logic errors as you read, great! If not, don't worry—that's what the debugging section of this chapter is for!

Designing the Script

Before you fire up Notepad or your favorite script editor, you need to sit down and figure out exactly what your script will do. This is the best way to answer the question "Where do I start?", which is the most common question you'll have when you start writing your own administrative scripts. By following a specific script design process like the one I'm about to show you, you'll always know exactly where to start, and the script itself will come much easier when you start programming.

Whenever I design a script, I use a three-step process.

1. Gather facts.

   This step lets me document what I know about my environment that will affect the script. I'm simply writing down the various things that my script will need to know, or that I'll need to consider as I write the script. This may include details about how Windows works, specific business requirements, and so forth.

2. Define tasks.

   This step lets me define the specific tasks my script will accomplish. I get detailed here, focusing on each tiny step I'd have to perform if I were manually performing what I want my script to do.

3. Outline the script.

   This step rolls up what I know and what I want to do into a sort of plain-English version of the script. I list each step I think the script will need to take, along with any related information. This becomes the basis for the script I'll write, and scripting itself becomes a simple matter of translating English into VBScript.

In the next three sections, I'll go through this design process with the IIS log rotation tool that you'll be helping me develop in this chapter. If you'd like to practice, take a few moments and walk through the process yourself before reading my results in the following sections.

Gathering Facts

What do you know about IIS and log files? You need to capture the information that your script will need to operate, such as log file locations, names, and so forth. After giving it some thought, I come up with the following list.

- Filenames. IIS log files use a file naming format that's based upon the date. Each log filename starts with the letters "ex," followed by a two-digit year, a two-digit month, and a two-digit day. The log file uses the filename extension .log.
- Files are stored in C:\Winnt\System32\LogFiles by default, at least on a Windows 2000 system. Windows Server 2003 uses C:\Windows\System32\LogFiles.
- I can store my archived files anywhere I want, so I'll create a folder named C:\Winnt\LogArchive. I'm assuming a Windows 2000 Server computer; for Windows Server 2003, I'd probably use C:\Windows\LogArchive instead.
- IIS closes each log file at the end of the day and opens a new one. I probably shouldn't try to move the log file that's currently opened by IIS; I should just go for yesterday's log file, instead.

Writing Functions and Subroutines

Generally, any kind of subtask you've identified is a great candidate for a function or subroutine, because subtasks get used more than once. You'll need to carefully examine your subtasks and decide which ones should be written as functions or subroutines. I have a general rule that I use: If a subtask involves more than one line of VBScript to accomplish, I write it as a function or subroutine. If I can do it in one line of VBScript code, I don't bother with a separate function or subroutine.

If you need a quick refresher of functions and subroutines, flip back to "What Are Functions?" in Chapter 5.

Identifying Candidate Modules

In this log rotation tool, I've already identified two potential modules (functions or subroutines): The date calculation and the log filename bit. A quick read through the VBScript documentation leads me to the DateAdd function, which can be used to calculate past or future dates. That seems to cover the date calculation subtask, so I don't think I'll need to write a function for that. I do see several Format commands that will help format a log filename, but none of them seem to do everything that I need in one line of code (at least, not one reasonably short line of code); I'll write the filename formatter as its own module.

Writing the Filename Formatting Function

Before writing a function, I need to consider a couple of facts. One fact is that the function is designed to encapsulate some subtask. Therefore, the function is going to need some kind of input to work on, and it's going to give me back some result that my script needs. Defining that input and output is critical. I want the function to be generic enough to be reusable, but specific enough to be useful.

Defining Function Input

In the case of the filename formatter, I know that the filename is always going to start with "ex," so I don't need that information in the input. The filename will always end in .log, so I don't need that in the input, either. What changes from filename to filename is the date information, so that seems like a logical piece of information for the function's input.

Defining Function Output

I want this function to take a date—its input—and create a fully formatted log filename. The output is obvious: a fully formatted log filename.

Writing the Function

Writing the actual function code requires a bit more task definition. You need to really break the task of formatting a filename down into small pieces. This can be a tough process, because the human brain does so many things for you without conscious thought. Think about what a three-year old would have to do to accomplish this task: Remember, all they have to work with at the beginning is a date.

You might come up with a task list like this.

1. Start with a blank piece of paper.
2. Write "ex" on the piece of paper.
3. On a separate piece of paper, write down the date you were given.
4.

Writing the Main Script

Now you're ready to fire up your script editor and write the main portion of the script. Any functions or subroutines you've written—including the FormatLogFileName function—will need to be copied and pasted into the first pat of the script.

NOTE

You can add the function to the script at the end, if you want. It's strictly a matter of personal preference.

➤➤ Log Rotation Script

With the supporting functions out of the way, you can start concentrating on the main script. Refer back to your original task list and translate it to VBScript; you might come up with something like Listing 13.2.

Listing 13.2. Log Rotation.vbs**. This is the first-pass script and contains all the important program logic.**

```
' Sample log rotation tool

'

' We'll take yesterday's log and move it to

' an archive folder. We'll delete the log file

' that's 30 days old from the archive


' ------------------------------------------------------------

'declare variables

Dim sLogPath, sService, sArchive, sLogFile

Dim oFSO

Dim d30Days, dYesterday




' ------------------------------------------------------------

' set up variables for folder locations

sLogPath = "c:\winnt\system32\logfiles\"

sService = "w3svc2\"

sArchive = "c:\winnt\LogArchive\"




' ------------------------------------------------------------

' get yesterday's date
```

Testing the Script

You're ready to test your script. Just to make sure you're on the same page, Listing 13.4 lists the entire log rotation script, including the FormatLogFileName function.

Listing 13.4. LogRotation3.vbs. **Here's the entire script, ready to run.**

```
' Sample log rotation tool
'
' We'll take yesterday's log and move it to
' an archive folder. We'll delete the log file
' that's 30 days old from the archive


Function FormatLogFileName(dDate)


 Dim sFileName
 sFileName = "ex"


 Dim sYear
 sYear = DatePart("yyyy",dDate)
 sYear = Right(sYear,2)
 sFileName = sFileName & sYear


 Dim sMonth
 sMonth = DatePart("m",dDate)
 If Len(sMonth) = 1 Then
  sMonth = "0" & sMonth
 End If
 sFileName = sFileName & sMonth


 Dim sDay
 sDay = DatePart("d",dDate)
 If Len(sDay) = 1 Then
  sDay = "0" & sDay
 End If
 sFileName = sFileName & sDay
```

Review

In this chapter, you combined what you've learned about script design, VBScript basics, and the Windows FileSystemObject to create a completely functional tool for rotating IIS log files. I deliberately designed some errors into the first revision of the script to walk you through the debugging process, and I showed you some great tips for easily debugging scripts even without the Microsoft Script Debugger or other fancy tools.

You practiced a couple of key tasks in this chapter. The design process is very important, as it helps you gather facts about what your script needs to accomplish and figure out how to break those tasks down into scriptable steps. The debugging process is also very important, and you'll find that the techniques you practiced in this chapter will come in handy as you start developing your own administrative scripts.

COMING UP

You've finished with your VBScript crash course. If you'd like to start using advanced administration technologies like Active Directory Services Interface and Windows Management Instrumentation, head on to Chapter 14. If you want to start working on a Web-based administrative script, turn to Chapter 21. Finally, if you'd like to move on to advanced scripting concepts like security, start with Chapter 25.

Part III: Windows Management Instrumentation and Active Directory Services Interface

Chapter 14. Working with ADSI Providers

IN THIS CHAPTER

ADSI is the way to manipulate domain and local users, groups, and other domain objects. I'll show you how ADSI's collection of providers make a number of different directories—including NT and AD—easily accessible from within your scripts.

Using ADSI Objects

ADSI, the Active Directory Services Interface, is an object library very similar in nature to the FileSystemObject and WScript objects I covered in Chapters 11 and 12. ADSI is a bit more complicated than the objects you've worked with so far, mainly because the information ADSI deals with is inherently more complicated.

For example, with the FileSystemObject, you learned to use CreateObject to have VBScript load the object's DLL into memory and provide a reference to your script. For example:

```
Dim oFSO

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

That's not quite how you'll use ADSI, though. For example, to have ADSI change password policy in a domain named BRAINCORE, you'd use the following code.

```
Set objDomain = GetObject("WinNT://BRAINCORE")


objDomain.Put "MinPasswordLength", 8

objDomain.Put "MinPasswordAge", 10

objDomain.Put "MaxPasswordAge", 45

objDomain.Put "MaxBadPasswordsAllowed", 3

objDomain.Put "PasswordHistoryLength", 8

objDomain.Put "AutoUnlockInterval", 30000

objDomain.Put "LockoutObservationInterval", 30000

objDomain.SetInfo
```

Notice that the GetObject statement is used, rather than CreateObject. I like to remember the difference by telling myself that I'm not trying to create a domain, just get to an existing one. Another important part of that statement is WinNT://, which tells ADSI which provider to use. The two main providers you'll work with are WinNT: and LDAP.

NOTE

ADSI provider names are case-sensitive, so be sure you're using WinNT and not winnt or some other derivation.

The WinNT provider can connect to any NT-compatible domain, including AD. Obviously, the provider cannot work with advanced AD functionality like organizational units (OUs), which don't exist in NT domains. The WinNT provider can also connect to the local SAM and other services on member and standalone computers. The LDAP provider can connect to any LDAP-compatible directory, such as the Exchange 5.5 directory or Active Directory. Both providers can be used to obtain a reference to an entire domain, an OU (in AD), users, groups, and much, much more. You'll even find areas of functionality that overlap with Windows Management Instrumentation (WMI); that's because ADSI is a bit older, and when WMI came on the scene, it started taking over. In fact, it's possible that someday ADSI will fade away entirely and that WMI will become the single means of accessing management information. For now, though, there's plenty that ADSI can do that WMI cannot.

Using the WinNT Provider

With Active Directory several years old, and now available in its second version (Wind2003), why would you bother using the WinNT provider? Ease of use. Although the WinNT provider is definitely less functional than the LDAP provider is, it's easier to use, and there are certain functions that you cannot easily do with the LDAP provider, such as connecting to a file server service. You can do some of those things with WMI, but again…ease of use. There are just some things, as you'll see, that the WinNT provider makes easy. For example, in Chapter 10, I showed you how the WinNT provider can be used to connect to a file server and find out which users have a particular file open.

Here's an example of how the WinNT provider can be used to connect to a file server and list its available shares.

```
ServerName = InputBox("Enter name of server " & _

 "to list shares for.")



set fs = GetObject("WinNT://" & ServerName & _

 "/LanmanServer,FileService")

For Each sh In fs

     'do something with the share

Next
```

You can do the same thing in WMI.

```
'get server name

strComputer = InputBox("Server name?")



'connect to WMI

Set objWMIService = GetObject("winmgmts:" & _

 "\\" & strComputer & "\root\cimv2")



'retrieve the list of shares

Set colShares = objWMIService.ExecQuery _

 ("SELECT * FROM Win32_Share WHERE " & _

 "Type = 0")



'for each share returned...

For Each objShare In colShares

 'do something with the share

Next
```

Using the LDAP Provider

The ADSI LDAP provider looks superficially similar to the WinNT provider, but uses LDAP-style naming conventions to name specific objects. A typical LDAP connection might look like this:

```
Dim objDomain

Set objDomain = GetObject("LDAP://dc=braincore,dc-net")
```

Notice that the LDAP provider is specified, and then an LDAP naming path is listed. In this case, objDomain will become a reference to the braincore.net domain. Perhaps the most confusing part of these LDAP paths is figuring out which components to use.

- Use DC when specifying any portion of a domain name. Always list the domain name components in their regular order. For example, a domain named east.braincore.net would have an LDAP path of "dc=east,dc=braincore,dc=net". DC stands for domain component, not domain controller; this type of LDAP path will force ADSI to find a domain controller following Windows' normal rules for doing so.
- Use OU when specifying an organizational unit. For example, to connect to the Sales OU in the braincore.net domain, specify "ou=sales,dc=braincore,dc=net". Notice that the domain name components are still required, so that ADSI can locate the domain that contains the OU.
- Use CN when specifying a common name, such as a user, group, or any of the built-in AD containers. Remember that the Users, Computers, and Built-in containers aren't technically OUs, and so they can't be accessed with the OU component. To connect to the Users container, use "cn=Users,dc=braincore,dc=net". To connect to a specific user, you can just specify the user and domain name: "cn=Donj,dc=braincore,dc=net". You don't need to specify the OU, because AD won't normally allow two users in the same domain to have the same name.

NOTE

It doesn't hurt to specify the OU containing a user or group; in fact, with some LDAP directories, it's required. Even though you don't have to, try to get into the habit of using fully qualified domain names, such as "cn=DonJ,ou=Sales,dc=braincore,dc=net".

After you've bound to an object, you can work with its properties. For example, suppose I want to modify the description of a particular user group. The following code will do it.

```
Dim objGroup

Set objGroup = GetObject( _

 "cn=Sales,ou=EastSales,dc=domain,dc=com")

objGroup.Put "description", "Eastern Sales representatives"

objGroup.SetInfo
```

The Put method allows me to specify a property to modify (in this case, the description of the group), and a new

Other Providers

ADSI doesn't stop with LDAP and WinNT. Here are some of the other providers that you can work with.

- GC. This provider allows you to work with the Global Catalog on AD domain controllers that host a replica of the Global Catalog. It works similarly to the LDAP provider, but uses the TCP ports assigned to access the Global Catalog.
- OLE DB. This provider allows you to perform search operations on AD by using Microsoft's OLE DB database interface.
- IIS. Provides access to the IIS metabase, which contains all of IIS' configuration information.
- NDS. This provides connects to Novell NetWare Directory Services. Note that later versions of NDS also support LDAP queries, meaning you can use the more generic LDAP provider for some operations.
- NWCOMPAT. Connects to Novell NetWare Bindery directories, found in NetWare 3.x and later.

Because most of your administrative tasks will involve the LDAP and WinNT providers, I'm not going to provide coverage or examples of how to use these other ADSI providers. However, you can access the ADSI documentation online at msdn.microsoft.com/library to learn more about them, if necessary.

Review

With this brief introduction to ADSI out of the way, you're ready to start managing domains, users, and groups by writing scripts that incorporate ADSI. You've learned how to write ADSI scripts that utilize both the WinNT and LDAP ADSI providers, and you've learned a bit about how the two providers function. Remember that the WinNT provider is not limited just to NT domains; it works fine in AD domains, and also provides a way to work with the local SAM and services on standalone and member computers, including NT-based client computers.

COMING UP

Ready to start ADSI scripting? In the next chapter, you'll learn how to work with domains, including creating OUs and other objects at the domain level. In Chapter 16, you'll learn how to manipulate users and groups in a domain, using both the LDAP and WinNT providers.

Chapter 15. Manipulating Domains

IN THIS CHAPTER

You'll learn to work with OUs, domain settings, domain information, and other domain-related items. These items are the basis for ADSI domain management, and I'll provide plenty of examples to help get you started.

Working with domains via ADSI is often easier if you start at the top level. In the last chapter, you learned how to use both the WinNT and LDAP ADSI providers to get an object reference to the domain.

```
Dim objNTDomain, objADDomain

objNTDomain = GetObject("WinNT://DOMAIN")

objADDomain = GetObject("LDAP://dc=domain,dc=com")
```

After you have a reference to the domain, you can start working with its properties. That'll be the focus of the first part of this chapter; toward the end of this chapter, I'll show you how to work with the main domain-level objects, organizational units (OUs), by using the LDAP provider.

Obviously, you need to make sure you have ADSI running on your computer in order to use it. ADSI comes with Windows 2000 and Windows XP, as well as Windows Server 2003. It's available for, but not included with, Windows NT, Windows 95, Windows 98, and Windows Me. To install ADSI, simply install the Microsoft Directory Services client on these older operating systems. You can also visit the ADSI link located at www.microsoft.com/windows/reskits/webresources.

Querying Domain Information

Querying domain information by using the LDAP provider is easy. Connect to the domain and simply use the Get method, along with the desired attribute name.

```
Dim objDomain

objDomain = GetObject("LDAP://dc=domain,dc=com")

WScript.Echo objDomain.Get("minPwdAge")
```

Of course, you need to know the attribute names that you want to query. Some of the interesting domain LDAP attributes include

- pwdHistoryLength. The number of old passwords the domain remembers for each user.
- minPwdLength. The minimum number of characters per user password.
- minPwdAge. The minimum number of days a user must keep his password.
- maxPwdAge. Maximum number of days a user may keep his password.
- lockoutThreshold. The number of tries you have to guess a password before the account is locked out.
- lockoutDuration. How long a password is left locked out.
- lockOutObservationWindow. The time window during which the lockoutThreshold number of wrong password attempts will cause an account lockout.
- forceLogoff. Forces account logoff when account restriction time expires.

You can explore more of the domain's attributes by examining the domain and domainPolicy classes in the AD schema; I'll describe how to view the attributes associated with a class later in this chapter.

Querying this information by using the WinNT provider is remarkably similar, although the attributes' names do change somewhat. Here's an example.

```
Dim objDomain

objDomain = GetObject("WinNT://DOMAIN")

WScript.Echo objDomain.Get("MinPasswordAge")
```

As you can see, the syntax is virtually identical, with the ADSI connection string and the attribute name being the only differences.

Changing Domain Settings

In the last chapter, I showed you an example of how you can use the WinNT provider to modify a domain's password and lockout policies. Here it is again.

```
' first bind to the domain

set objDomain = GetObject("WinNT://MyDomain")


objDomain.Put "MinPasswordLength", 8

objDomain.Put "MinPasswordAge", 10

objDomain.Put "MaxPasswordAge", 45

objDomain.Put "PasswordHistoryLength", 8

objDomain.Put "LockoutObservationInterval", 30000

objDomain.SetInfo
```

This same syntax works pretty well for LDAP connections to a domain, although as I noted in the previous section the attribute names are different. Here's an LDAP version of the same example.

```
' first bind to the domain

set objDomain = GetObject("LDAP://dc=domain,dc=com")


objDomain.Put "minPwdLength", 8

objDomain.Put "minPwdAge", 10

objDomain.Put "maxPwdAge", 45

objDomain.Put "pwdHistoryLength", 8

objDomain.Put "lockoutObservationWindow", 30000

objDomain.SetInfo
```

As you can see, the basic syntax is to use the Put method, the appropriate attribute name, and the new value, and then to call the SetInfo method when you're finished. SetInfo copies the changes back to the directory, committing the changes.

Working with OUs

You'll likely do four basic things with an OU. By the way, some of these operations also apply to the built-in, OU-like containers: Users, Computers, and Built-In. Keep in mind that these are not proper OUs and cannot be accessed in quite the same way as I described in the previous chapter. In the next four sections, I'll demonstrate how to use ADSI to create, modify, query, and delete an OU.

NOTE

Because OUs don't exist in NT domains, all of these examples will only use the LDAP provider that works with Active Directory in its native mode.

Creating an OU

Creating an OU is simple enough. First, you need to obtain a reference to the parent of the new OU, and then use that object's Create method to create a new OU. To create a new top-level OU named Sales:

```
Dim objDomain, objNewOU

Set objDomain = GetObject("LDAP://dc=domain,dc=com")

Set objNewOU = objDomain.Create("organizationalUnit", "ou=Sales")

objNewOU.SetInfo
```

Classes and Attributes

As you're working with AD, it's important to understand the system of classes and attributes that the AD schema uses for its organization. An attribute is some discrete piece of information, such as a name or description. A class is simply a collection of attributes that describes some real-world object. For example, a user is a class that includes attributes such as name, description, address, and so forth. A group is another class, which includes such attributes as name, description, and members.

AD does not allow multiple attributes to use the same name. So, when you see two classes with the same attributes (such as description), both classes are actually using the same attribute definition from the AD schema. This sort of re-use makes AD very efficient.

An instance is a copy of a class with its attributes' values filled in. For example, DonJ might be the name of a particular user. The user object you see in the AD GUI is an instance of the user class.

Notice that the Create method returns a reference to the newly created object, and I still have to call that object's SetInfo method to save the changes into the directory. I could also modify properties of the new OU prior to calling SetInfo. Let me extend this example and create both a top-level Sales OU and a child OU named West under that.

```
Dim objDomain, objNewOU

Set objDomain = GetObject("LDAP://dc=domain,dc=com")

Set objNewOU = objDomain.Create("organizationalUnit", _

 "ou=Sales")
```

Putting It All Together

One potential use for domain- and OU-manipulation scripts is to configure a test or pilot domain that resembles your production domain. By using a script, you can install a domain controller in a lab, and then quickly recreate aspects of your production environment, such as OU structure and user accounts.

➤➤ Preload Domain

Listing 15.1 shows a script that preloads a domain with a specific OU structure. Just for fun, I've thrown in a couple of new methods that copy and move OUs around within the domain. See if you can figure out how they work before you read the line-by-line explanation.

Listing 15.1. PreLoad.vbs. **Preloads a specific OU configuration into a domain via LDAP.**

```
'bind to domain

Dim oDomain

Set oDomain = GetObject("LDAP://dc=domain,dc=com")


'Create top-level OUs

Dim oSales, oHR, oMIS

Set oSales = oDomain.Create("organizationalUnit", "Sales")

Set oHR = oDomain.Create("organizationalUnit", "HR")

Set oMIS = oDomain.Create("organizationalUnit", "MIS")

oDomain.SetInfo


'set descriptions

oSales.Put "description", "Sales OU"

oHR.Put "description", "HR OU"

oMIS.Put "description", "MIS OU"

'save

oSales.SetInfo

oHR.SetInfo

oMIS.SetInfo


'create child OUs for Sales

Dim oChild

Set oChild = oSales.Create("organizationalUnit", "Widgets")

oChild.SetInfo

Set oChild = oSales.Create("organizationalUnit", "Widgets")
```

Review

ADSI makes it easy to connect to and manipulate domains. You've seen how to query and modify domain-level attributes, and how to create, modify, query, and delete domain-level objects, such as OUs. These techniques can be applied not only to OUs, but also to users and groups, as you'll see in the next chapter. Having the ability to easily manipulate domain and OU information from script can allow you to restructure domains, automate bulk domain configuration tasks, and much more.

COMING UP

In the next chapter, I'll dive a bit deeper into ADSI and show you how to create, modify, and delete users and groups. I'll also show you tricks for querying ADSI, such as determining whether a user is a member of a particular group.

Chapter 16. Manipulating Users and Groups

IN THIS CHAPTER

With domains and OUs under your belt, you're ready to start writing scripts that manipulate and query the users and groups in your domains. I'll focus on using the LDAP provider for domain operations, and the WinNT provider for working with computers' local SAMs.

User and group maintenance is probably one of the top administrative tasks that you wanted to automate when you picked up this book. You may be interested primarily in domain user and group management, or local computer user and group management, or possibly both. Remember that the WinNT ADSI provider can be used both in NT domains and, for limited operations, in Active Directory (AD) domains. The WinNT provider also gives you access to the SAM on standalone and member servers and NT-based client computers, such as Windows XP machines. The LDAP provider is AD's native provider, and gives you the best access to AD's capabilities, including the ability to work with OUs.

In an AD domain, the WinNT provider gives you a flat view of the domain: All users are in a single space, not separated into containers and OUs. With the LDAP provider, however, you need to remain aware of your domain's OU structure, and you need to become accustomed to fully qualified domain names (FQDNs) that describe users and groups not only by their name, but also by their position within the domain's OU hierarchy.

Creating Users and Groups

Creating users and groups is probably one of the most frequently automated tasks for administrators, or at least the task they'd most like to automate. Scripting makes it easy, whether you're using the WinNT provider or the LDAP provider.

The WinNT Way

With the WinNT provider, you start by obtaining a connection to the domain itself. Because all users and groups exist at the top level of the domain, you don't need to connect to a specific OU. Note that you can also use this technique to create local user and group accounts, by simply connecting directly to a non-domain controller instead of connecting to a domain.

TIP

If you want to create a user or group on a specific domain controller, thus making it available immediately on that domain controller without waiting for replication to occur, connect to the domain controller by name rather than connecting to the domain. Domain controllers don't technically have local accounts, so when you attempt to create new local accounts on a domain controller, you're really creating domain accounts.

After you are connected, simply use the Create method—much as I did with OUs in the previous chapter—to create the user account. Here's an example.

```
Dim oDomain, oUser

Set oDomain = GetObject("WinNT://DOMAIN")

Set oUser = oDomain.Create("user","DonJ")
```

Not much to it. You need to call SetInfo to save the new user, but first you probably want to set some of the user's attributes. Here's an extended example.

```
Dim oDomain, oUser

Set oDomain = GetObject("WinNT://DOMAIN")

Set oUser = oDomain.Create("user","DonJ")


oUser.SetPassword "pa55w0rd!"

oUser.FullName = "Don Jones"

oUser.Description = "Author"

oUser.HomeDirectory = "\\server1\donj"

oUser.RasPermissions = 9

oUser.SetInfo
```

The WinNT provider helpfully exposes these attributes as properties of the user object, meaning you don't have to use raw attribute names like you do with the LDAP provider (which I'll cover next).

Querying User Information

Reading user information (or group information, for that matter) requires the use of the Get method, as well as the name of the attribute you want to read. In the previous chapter, I showed you how to use the AD Schema console to browse a class for its available attributes; you can use the same technique on the user and group classes to see what attributes they support. To query information, simply connect to the object in question and use Get to retrieve the attribute values that you need.

```
Dim oUser

Set oUser = GetObject("LDAP://cn=DonJ,ou=MIS,dc=domain,dc=com")

WScript.Echo oUser.Get("name")

WScript.Echo oUser.Get("description")

WScript.Echo oUser.Get("sAMAccountName")
```

That's easy enough. Using the WinNT provider, you can directly access many attributes that are exposed as regular properties.

```
Dim oUser

Set oUser = GetObject("WinNT://DOMAIN/DonJ")

WScript.Echo oUser.Name

WScript.Echo oUser.Description
```

One thing to be careful of with the WinNT provider is that it grabs the first object it finds matching your query. For example, if I have a user and a group named DonJ, the preceding example might bind to the user or the group. You can force the object type by specifying it.

```
Dim oUser

Set oUser = GetObject("WinNT://DOMAIN/DonJ,user")

WScript.Echo oUser.Name

WScript.Echo oUser.Description
```

You can also use Get with the WinNT provider, making its syntax parallel to the LDAP provider. Keep in mind that user objects have a number of multivalued attributes, as I mentioned in Chapter 14. Reading those requires a slightly different technique.

```
Dim oUser

Set oUser = GetObject("LDAP://cn=DonJ,ou=MIS,dc=domain,dc=com")


Dim sURL
```

Changing User Settings

Using the LDAP provider, you can use Put to change user and group attributes.

```
Dim oUser

Set oUser = GetObject("LDAP://cn=DonJ,ou=MIS,dc=domain,dc=com")

oUser.Put "description", "Author"

oUser.SetInfo
```

Keep in mind that users in particular offer a number of multivalued attributes. I discussed how to work with those in Chapter 14. Here's quick refresher.

```
Const MVP_CLEAR = 1

Const MVP_UPDATE = 2

Const MVP_APPEND = 3

Const MVP_DELETE = 4


Dim objUser

Set objUser = GetObject("cn=DonJ,ou=Sales,dc=braincore,dc=net")

objUser.PutEx MVP_APPEND, "otherTelephone", Array("555-1212")

objUser.SetInfo
```

This example appends another telephone number to a user's otherTelephone multivalued attribute. You can also clear the attribute completely, delete entries, or change a particular entry. The following example adds a new telephone number, and then deletes it.

```
Const MVP_CLEAR = 1

Const MVP_UPDATE = 2

Const MVP_APPEND = 3

Const MVP_DELETE = 4


Dim objUser

Set objUser = GetObject("cn=DonJ,ou=Sales,dc=braincore,dc=net")

objUser.PutEx MVP_APPEND, "otherTelephone", Array("555-1212")

objUser.SetInfo
```

Working with Groups

You'll want to do two primary things with groups: modify their membership and check their membership. The former can be useful in scripts that bulk-add new users to the domain; the latter is invaluable in logon scripts. Let's take checking group membership first. The basic trick is to get a reference to a group, and then scan through its members until you find a particular user (or not). This is best implemented as a function, which can be easily reused in different scripts. The function is in Listing 16.1.

Listing 16.1. CheckGroupMembership.vbs. **This function checks to see if a specified user belongs to a specified group.**

```
Function IsMember(sUser, sGroup)

 Dim oGroup, bIsMember, oMember

 bIsMember = False

 Set oGroup = GetObject("LDAP://" & sGroup)

 For Each oMember in oGroup.GetEx("member")

  If oMember.Name = sUser Then

   bIsMember = True

   Exit For

  End If

 Next

 IsMember = bIsMember

End Function
```

You need to pass FQDNs to this function. For example, to see if user DonJ, located in the MIS OU, is a member of the HelpDesk group, also located in the MIS OU, you'd do something like this.

```
If IsMember( _

 "cn=DonJ,ou=MIS,dc=domain,dc=com", _

 "cn=HelpDesk,ou=MIS,dc=domain,dc=com") Then

 WScript.Echo "He's a member!"

Else

 WScript.Echo "He's not a member!"

End If
```

Notice that the function uses the GetEx method to retrieve the group object's member attribute, which is a multivalued attribute. Each entry in the attribute is the FQDN of a user that belongs to the group. The benefit of a function like this is that it can check for users from different domains belonging to, for example, a Universal security group, because you're using the FQDN of the user, which includes his home domain.

Putting It All Together

In the previous chapter, I demonstrated a script that sets up a domain with some OUs, designed to model a production environment in a test lab. But what's a domain without users?

➤➤ Preload Domain II

Listing 16.2 shows a script that utilizes everything I've covered in this chapter. It's designed to be added to the end of Listing 15.1 for a complete domain preloading script. This script creates a couple of thousand users accounts, some groups, and distributes users into the groups.

Listing 16.2. PreloadDomain2.vbs. **Creating dummy user and group accounts for a domain in a test environment.**

```
'create 10,000 user accounts
'seriously - don't run this in a
'production domain!


'connect to the root
Dim oRoot
Set oRoot = GetObject("LDAP://rootDES")


'connect to the Users container
Dim oContainer
Set oContainer = GetObject("LDAP://cn=Users," & _
 oRoot.Get("defaultNamingContext")


'create 10,000 users
Dim iUser, oUser
For iUser = 1 To 10000
 Set oUser = oContainer.Create("user", _
  "DummyUser" & CStr(iUser)
 oUser.SetInfo
Next


'create 1,000 groups
Dim iGroup, oGroup
For iGroup = 1 To 1000
 Set oGroup = oContainer.Create("group"
```

Review

Working with users and groups is relatively easy from within ADSI. Remember that you can use the WinNT provider to access not only Windows NT domains, but also Active Directory domains, standalone computers, domain member computers, and so forth. Native Active Directory access is provided through the LDAP provider, which also provides access to other LDAP-based directories, such as Exchange 5.x. Some of the most useful scripts you'll develop will use ADSI to manage local user accounts, such as service accounts and built-in accounts like Administrator.

COMING UP

You're ready to start working with Windows Management Instrumentation (WMI), Microsoft's way of providing you with almost total administrative control of your computers through scripts. WMI builds upon the object-based scripting you've been using for files and directory services, and provides a nearly unlimited range of administrative possibilities.

Chapter 17. Understanding WMI

IN THIS CHAPTER

You've no doubt heard about Windows Management Instrumentation and how it's the holy grail of systems administration. You may have even looked into it and realized how complicated it appears to be! WMI is a powerful tool, but I'll show you that it's not as complicated as it appears.

Whenever I speak at conferences, I'm nearly always asked about Windows Management Instrumentation, or WMI. WMI first caught on in Windows 2000 (although it's partially supported in Windows NT 4.0), and administrators have been hearing about how wonderful a tool it is for managing systems, especially through scripting. Unfortunately, WMI is also one of the most complex-looking technologies to have come out of Redmond in a long time, and many administrators are justifiably concerned about having to spend the rest of their lives understanding it. In this chapter and the two that follow, however, I'm going to show you that WMI isn't as complicated as it looks. In fact, I'll even provide you with some code templates that you can modify to query or set almost any kind of management information from a Windows computer.

The WMI Hierarchy

One of the most complicated parts of WMI is the sheer number of acronyms that come with it: DMTF, CIM, Win32, and so forth. First, bear in mind that you don't really need to remember any of them to use WMI effectively. However, it can be helpful to understand what they all mean, because they help WMI make more sense.

The DMTF is the Desktop Management Task Force. It's an industry group primarily concerned with making desktop computers (they do care about servers, too) easier to manage. Microsoft pays close attention to the DMTF and is a contributing member. One of the things that the DMTF realized is that every hardware, software, and operating system vendor has different names for the same things. Windows, for example, has logical disks, partitions, volumes, and so forth; Novell NetWare uses these terms for slightly different things. To clear up the confusion, the DMTF created the Common Information Model, or CIM.

The CIM is essentially a generic way of describing everything associated with a computer, at both a hardware and a software level. The CIM defines many base classes to represent things like disks, processors, motherboards, and so forth. The CIM classes only include properties that are universal. For example, the CIM_DiskDrive class includes a property for Name, because all disk drives can be assigned a descriptive name. It also includes a property for MaxBlockSize, because all disk drives manufactured today have an associated maximum block size. The class doesn't include a property that indicates the file system used to format the disk, nor does it show whether a disk is basic or dynamic. Those are operating-system-specific features not addressed by the CIM.

The CIM is, however, extensible. When Microsoft created WMI, it created its own series of Win32 classes that are Windows-specific. The Win32 classes are based on, or inherited from, CIM classes. For example, there's a Win32_DiskDrive class. It includes all of the properties associated with the CIM_DiskDrive class, and includes additional properties—such as PNPDeviceID—that are specific to the Windows operating system.

TIP

You might want to explore the WMI reference information online, just to see how the Win32 classes build upon their CIM counterparts. Go to http://msdn.microsoft.com/library to start. In the left-hand navigation tree, open Setup and System Administration, Windows Management Instrumentation, SDK Documentation, WMI Reference, and WMI Classes. You'll see sections for CIM classes and Win32 classes.

The main part of WMI is understanding that it's composed of these classes, which represent the hardware and software in a computer. My laptop, for example, has once instance of the Win32_DiskDrive class, which simply means that the machine contains one disk drive. My desktop machine has two instances of Win32_DiskDrive, which means it contains two hard disks. Absolutely everything in WMI is set up to handle multiple instances of classes. Sometimes, that doesn't seem to make any sense. After all, how many computers do you know of that contain multiple instances of a class like Win32_MotherboardDevice? Not many! But WMI is designed to be forward looking. Who knows; we might someday be working with computers that do have multiple motherboards, and so WMI is set up to deal with it.

Multiple instances can make querying WMI information seem complex. For example, suppose you want to query the IP address of a workstation's network adapter. Unfortunately, you cannot just ask for the IP address from the first adapter WMI knows about. Windows computers all contain multiple network adapters, if you stop to consider virtual VPN adapters, the virtual loopback adapter, and so forth. So, when you write WMI queries, you have to take into account the fact that the computer probably contains multiple instances of whatever you're after, and write your script accordingly. As a quick example, try the script in Listing 17.1.

Listing 17.1. ShowNIC.vbs. **Shows the IP address and MAC address of each network adapter you have.**

```
Dim strComputer

Dim objWMIService
```

Exploring WMI's Capabilities

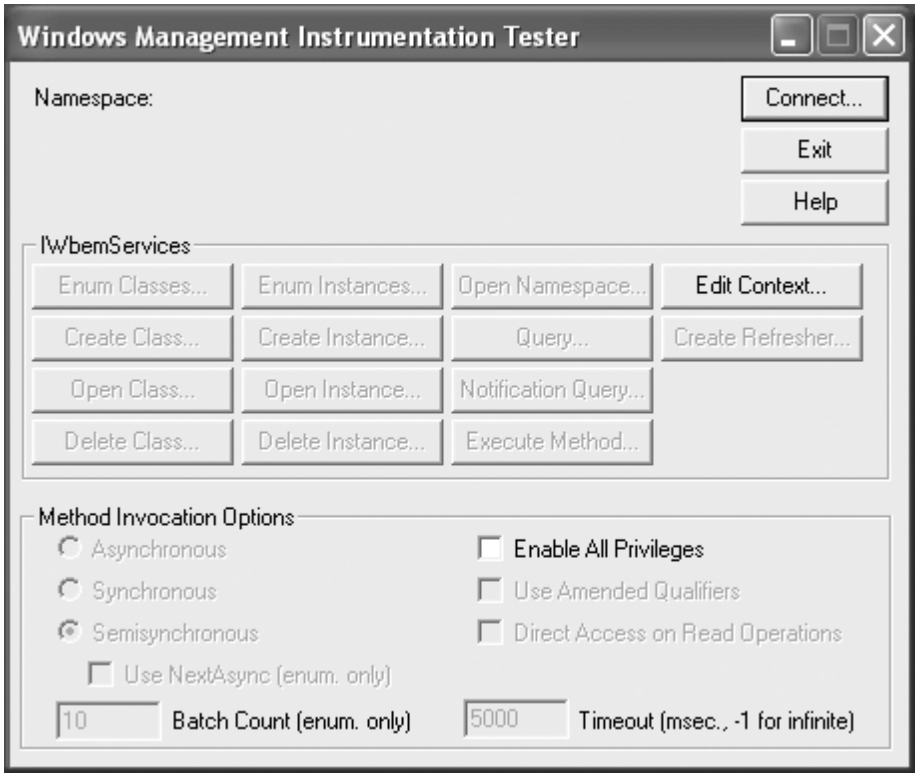
Perhaps the easiest way to understand WMI is to simply start playing with it. Windows XP and Windows Server 2003 include Wbemtest.exe, a tool that can be used to test WMI functionality and explore its capabilities.

NOTE

Another acronym! WBEM stands for Web-Based Enterprise Management, Microsoft's implementation of several key DMTF technologies that includes WMI. You don't see the WBEM name as much as you used to, but it still pops up in tool names and the like.

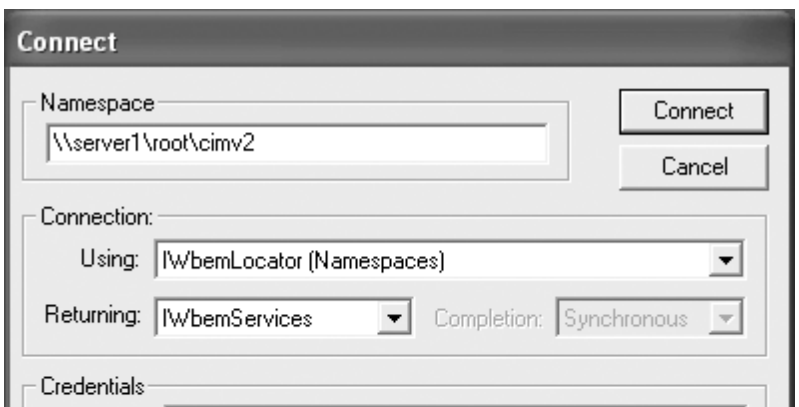
To run Wbemtest, simply select Run from the Start menu, type wbemtest, and click OK. You'll see the main Wbemtest panel, shown in Figure 17.1.

Figure 17.1. The WMI Tester's main window



The first thing you need to do is connect to a WMI provider. Generally, that means connecting to the Windows Management Instrumentation service on your local machine or on another computer. I like to connect to the one on another computer, because it demonstrates WMI's real power as a remote administration tool. To connect, click the Connect button. You'll see the Connect dialog box, shown in Figure 17.2

Figure 17.2. Connecting to a remote machine's WMI namespace



Installing WMI

As I mentioned earlier, WMI is preinstalled on Windows 2000 and all later Windows operating systems, including Windows Me. However, if you're using anything earlier, you may need to install WMI before you can start deploying WMI scripts. WMI must be installed on every computer that you intend to query, regardless of where your scripts will actually run; WMI must also be installed on any computer that will run WMI scripts. To obtain the WMI installer, go to the Microsoft home page and select Downloads. From the left-hand menu, select the System Management Tools category. Look for the Windows Management Instrumentation (WMI) CORE download for WMI version 1.5. Downloads are available for Windows 9x and NT 4.0. If you cannot spot the downloads in the list, simply type WMI into the keyword search at the bottom of the page.

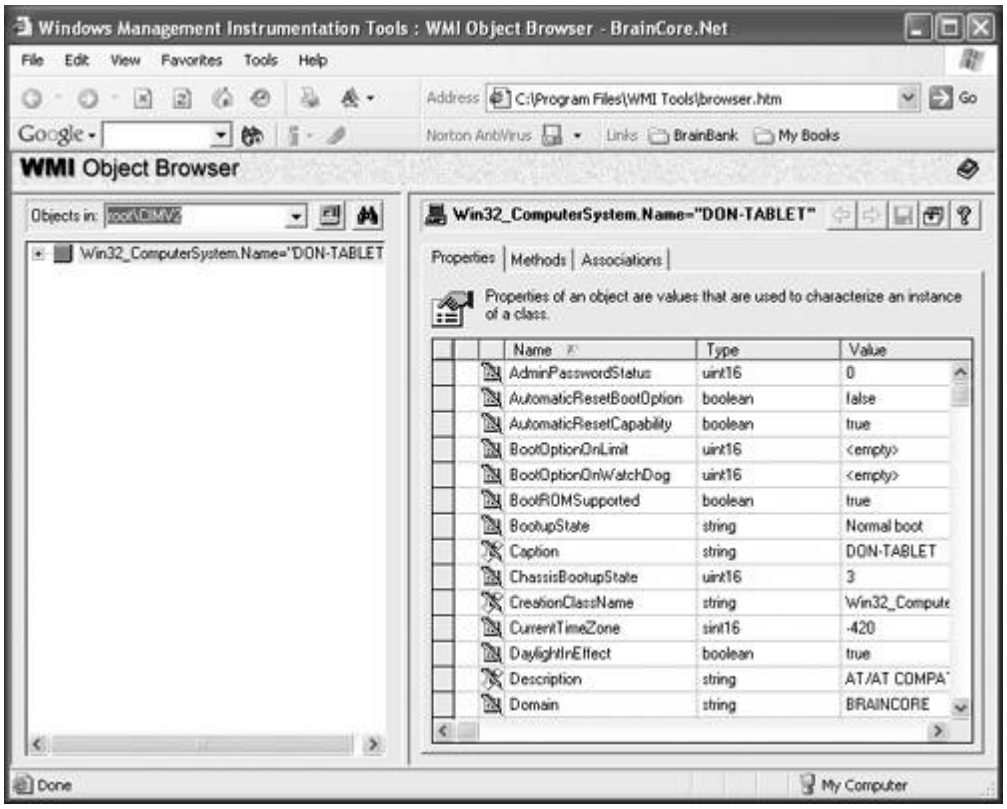
The installer is an executable, not an MSI package. Unfortunately, because these older operating systems don't support Group Policy software deployment, you'll have to manually install the package, or deploy it through alternative means such as Microsoft Systems Management Server (SMS).

I also recommend that you download and install the WMI Administrative Tools. Again, go to the Microsoft home page and select Downloads. From the left-hand menu, select the System Management Tools category. Finally, look for WMI Administrative Tools in the list. I'll discuss the administrative tools in the next section.

Using the WMI Tools

I've already introduced you to Wbemtest, which is a great way to experiment with WMI and get a feel for what it can do. The WMI Administrative Tools, however, includes the WMI Object Browser, which is an exceptionally cool tool. After downloading and installing the tools, launch the Object Browser from the Start menu. Have it connect to the root/CIMV2 namespace, and provide logon credentials if necessary. You'll see the main screen, shown in Figure 17.7.

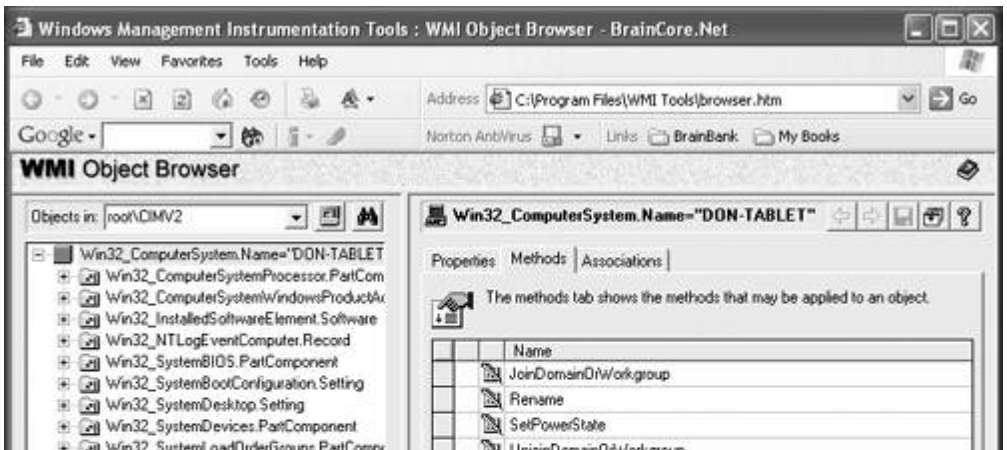
Figure 17.7. The main Object Browser screen



The Browser lets you see all the properties associated with each class. For example, it starts connected to the Win32_ComputerSystem class that represents your entire computer; you can see the properties of the class—such as AutomaticResetBootOption—that govern many aspects of your computer's behavior.

On the Object Browser's Methods tab, shown in Figure 17.8, you can see the actions that the class can perform. The Win32_ComputerSystem class, for example, offers a JoinDomainOrWorkgroup method, a Rename method, a SetPowerState method, and an UnjoinDomainOrWorkgroup method. These methods can be programmatically called from within your scripts (which I'll explore in the next two chapters), allowing you to change the computer's configuration.

Figure 17.8. Examining the methods of a WMI class



Really—It's This Easy

In my conference lectures on scripting, I always try to prove how easy WMI scripting really is. I usually ask students to call out some piece of computer information that they'd like to be able to query. Believe me, I haven't memorized the hundreds of WMI classes that are available, so it's unlikely that I'll already know how to query whatever they ask for. It's a great way to show how a little documentation and a couple of tools can quickly result in a powerful WMI script. For example, suppose you need to query a server to see if any persistent routes have been added by using the route –p add command. No problem. Here are the four steps to writing almost any WMI script.

Find the Class

First, I have to figure out what to query. This is easily the toughest part of the entire process. I usually start in the WMI Reference documentation, looking at the five categories of Win32 classes:

1. Computer System Hardware
2. Operating System
3. Installed Applications
4. WMI Service Management
5. Performance Counter

Of these five, Operating System seems to be the most likely choice for routing information, so I'll expand that topic. Unfortunately, that leaves me with a whole bunch of classes still to work through. Fortunately, they're alphabetical, so I can scroll right down to the R section and look for something like Win32_Route. Nope, nothing. In fact, Win32_Registry is the only thing under R, and that clearly isn't it.

Idly scrolling back up, I do see Win32_IP4RouteTable. Aha! That makes sense; Windows XP and Server 2003 both support IPv4 and IPv6; WMI clearly needs some way to distinguish between the two. Looking more closely, I also see Win32_IP4PersistedRouteTable, which looks exactly like what I want.

Here's what the Microsoft MSDN Library has to say about Win32_IP4PersistedRouteTable.

The Win32_IP4PersistedRouteTable WMI class represents persisted IP routes. By default, the routes added to the routing table are not permanent. Rebooting the computer clears the routes from the table. However, the following Windows NT command makes the route persist after the computer is restarted:

```
route -p add
```

Persistent entries are automatically reinserted in the route table each time the route table is rebuilt. Windows NT stores persistent routes in the registry. This class is only applicable to IP4 and does not address IPX or IP6. An entry can be removed through the method call SWbemServices.Delete (in the Scripting API for WMI) or IWbemServices::DeleteInstance (in the COM API for WMI). This class was added for Windows Server 2003 family.

That last sentence gives me some pause: "This class was added for Windows Server 2003 family." Scrolling to the

Review

WMI looks complex, but that's primarily because there's so darn much of it. Boiled down, WMI isn't difficult at all, and can really be a lot of fun when you get used to it. In this chapter, you've learned how WMI works, how you can access it from your scripts, and how to methodically create WMI scripts to perform almost any task. You also learned about some of the tools that make WMI easier to work with, such as Wbemtest and the WMI Object Browser. I also introduced you to the WMI Scriptomatic from Microsoft, which makes creating new WMI scripts a real breeze.

COMING UP

Now that you have the WMI basics under your belt, it's time to start querying and working with basic information. In the next chapter, I'll do just that, giving you plenty of examples to work with. Afterward, I'll move on to querying more complex information, and show you how easy it can be to make even complex WMI tasks scriptable.

Chapter 18. Querying Basic WMI Information

IN THIS CHAPTER

You've already learned how to perform very basic WMI queries; now it's time to really learn how those queries are working under the hood. I'll walk through the query-building process in more detail, and show you various ways to manipulate and work with the information you retrieve from WMI.

In the previous chapter, I showed you a standard template-style WMI query that you can modify to query almost anything from WMI. What I didn't do is show you exactly how that query works, how you can easily incorporate it into other scripts, and how to utilize the information you retrieve. If you start using WMI examples from the Web, you might even notice that different script authors write their WMI queries in completely different ways. There's nothing wrong with that, because WMI is flexible enough to work in different ways and still achieve the results you need.

The WMI Query Language (WQL)

The WMI Query Language, or WQL, is a subset of the industry-standard Structured Query Language (SQL) defined by the American National Standards Institute (ANSI). Although there are other ways to retrieve information from WMI, writing a WQL query is probably the easiest, because WQL closely resembles normal English syntax and grammar.

In the previous chapter, you saw examples of some basic WQL queries.

- SELECT * FROM Win32_NetworkAdapterConfiguration WHERE DHCPEnabled= TRUE
- SELECT Description FROM Win32_Account WHERE Name= 'Administrator'
- SELECT Freespace,DeviceID FROM Win32_LogicalDisk

Queries like these will likely be the ones you use most; however, it's useful to understand what else you can do with WQL, especially when working with complex information.

Regular SQL has literally hundreds of keywords and clauses; WQL, on the other hand, has 19. That's a much more manageable number, and it means you'll be able to master WQL without rivaling your company's database administrators in SQL prowess. Of course, if you already know SQL, WQL is going to be a snap.

Complex WMI Information

I've used the phrase complex information a couple of times in this and the previous chapter; the next chapter, in fact, has complex information right in the title. What does it mean?

I divide WMI information into two categories: simple and complex. Simple information is the kind that typically only has one instance on a computer. For example, if I want to query a computer's serial number, there's only going to be one of those. More complex information, like TCP/IP addresses, require more effort as a programmer, because each computer can have multiple network adapters, and each network adapter can have multiple addresses.

Security information can be even more complex. For example, WMI provides a way to access NTFS file permissions. Each file on the hard drive is an instance of a WMI class, and each user or group in the computer or domain is represented by a different class. In between those two classes are access control entries, or ACEs, which grant a specific permission on a specific file to a specific user or group. So, to access NTFS file permissions, you're dealing with at least three interrelated classes. Complex enough for you?

Properly written WQL queries can reduce this complexity by allowing you to query for specific sets of data, rather than having to wade through all the interrelated classes.

NOTE

I'm not going to cover all 19 keywords. Several of them are intended for querying WMI events, which are special notifications generated by WMI when specific things occur, as a file being modified. Dealing with WMI events is a bit beyond the scope of this book, and better suited to traditional programming than scripting.

Determining What to Query

I mentioned in the previous chapter that actually figuring out which WMI class to query is the truly tough part about working with WMI, and it's true. The Microsoft MSDN Library reference to the WMI classes, particularly Microsoft's Win32 classes, is the most comprehensive and useful place to start looking. The documentation can be found online at http://msdn.microsoft.com/library. Just expand Setup and System Administration in the left-hand menu, then navigate to Windows Management Instrumentation, SDK Documentation, WMI Reference, WMI Classes, and Win32Classes.

Microsoft currently divides the classes into five categories, although that will almost certainly change over time as the classes are expanded. The current categories are

- Computer System Hardware, which includes everything you can physically touch and see. This includes network adapters, the motherboard, ports, and so forth.
- Operating System, which includes everything associated with Windows itself: users and groups, file quotas, security settings, COM settings, and more.
- Installed Applications, which covers the Windows Installer subsystem and all managed applications.
- WMI Service Management, which covers the configuration and management of WMI itself.
- Performance Counter, which provides access to performance monitoring data through WMI.

After you've narrowed down the proper category, my best advice is to dive into the documentation and scroll through the class names until you find one that looks like it will do what you want. Need to force a hard drive to run CHKDSK? Hard drives are hardware, but CHKDSK is Windows-specific. After all, you don't really run CHKDSK on a hard drive, do you? You run it on a volume, which is a Windows thing. So, start with the Operating System category. There's the Win32_LogicalDisk category, which represents a volume like C: or D:. Lo and behold, it has a ChkDsk method and a ScheduleAutoChk method, one of which is sure to do the trick. The documentation also helpfully notes that the method is included only in Windows XP and Windows Server 2003, meaning that you'll have to find another way to handle earlier clients.

Microsoft's categorization of the WMI classes is far from consistent. For example, Win32_NetworkAdapterConfiguration is included in the Computer System Hardware category. Although I agree that a network adapter is definitely hardware, surely its actual configuration is part of the operating system, right? In other words, be prepared to do a little browsing to find the right classes, especially until you become accustomed to them.

TIP

The Appendix of this book is a Quick Script Reference I put together to help you locate the right WMI classes more quickly. For example, if you need to write a WMI query to retrieve Windows Product Activation information, just look up "Activation" in the Quick Script Reference. You'll see a reference to WMI, an indication that it's covered in Chapters 17 through 19, and the specific classes involved: Win32_WindowsProductActivation, for example.

No matter what, don't get discouraged. Keep browsing through the list until you find what you want. Just so you know, WMI isn't complete, yet. Microsoft hasn't provided a WMI "hook" for each thing Windows can do. In fact, the coverage for Windows XP and Windows Server 2003 is light-years better than what's in Windows 2000, and that's better still than NT. But even Windows Server 2003's implementation of WMI doesn't let you query or control the DHCP service, modify IPv6 in any way, modify DNS server records (although you can do that through ADSI in

Testing the Query

In the previous chapter, I showed you how to write and test a query using the Wbemtest tool. I recommend that you test every query you plan to write, by running Wbemtest on the target operating system. That way, you'll know your queries are returning the correct results before you spend a lot of time writing an actual script.

For specific instructions on testing a query, see "Really—It's This Easy" in Chapter 17.

If your script will run on multiple operating systems (as in a logon script or a script being used to manage multiple remote servers), be sure to test the query on each potential operating system. That way, you'll quickly spot any WMI version incompatibilities, and you can take the appropriate steps. Don't forget that you can also test your query by using a generic WMI query script, such as the kind generated by the WMI Scriptomatic or by PrimalScript's WMI Query Wizard.

For example, suppose I want to test the Win32_QuotaSetting query. By using PrimalScript, I just run the wizard, select Win32_QuotaSetting from the class list, and click Insert. The wizard creates the following script.

```
On Error Resume Next

Dim strComputer

Dim objWMIService

Dim colItems



strComputer = "."

Set objWMIService = GetObject( _

 "winmgmts:\\" & strComputer & "\root\cimv2")

Set colItems = objWMIService.ExecQuery( _

 "Select * from Win32_QuotaSetting",,48)

For Each objItem in colItems

 WScript.Echo "Caption: " & objItem.Caption

 WScript.Echo "DefaultLimit: " & objItem.DefaultLimit

 WScript.Echo "DefaultWarningLimit: " & _

  objItem.DefaultWarningLimit

 WScript.Echo "Description: " & objItem.Description

 WScript.Echo "ExceededNotification: " & _

  objItem.ExceededNotification

 WScript.Echo "SettingID: " & objItem.SettingID

 WScript.Echo "State: " & objItem.State

 WScript.Echo "VolumePath: " & objItem.VolumePath

 WScript.Echo "WarningExceededNotification: " &

  objItem.WarningExceededNotification
```

Writing the Query in VBScript

If you're like me, you like your final scripts to be clean, consistent, and easy to read. Using the Wizard- or Scriptomatic-generated scripts isn't the best way to achieve consistency. For example, the PrimalScript Wizard always includes the following code.

```
On Error Resume Next

Dim strComputer

Dim objWMIService

Dim colItems



strComputer = "."

Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\root\cimv2")
```

First, you might not want error-checking turned off, which is what On Error Resume Next does. You might use a different variable naming convention (I often do, mainly because I'm a bit too lazy to type str instead of just s for string variables and the like), or you might have already defined a variable name that the wizard is using. Understand that you can always revise and modify the template scripts to fit better within your overall scripts. Not only can you change them, you probably should change them.

Suppose you want to write a script that restarts a remote server. You've done your browsing, and Win32_OperatingSystem has a method named Shutdown that looks like it'll do the trick. Using Scriptomatic or the WMI Query Wizard, you generate code similar to the following.

```
On Error Resume Next

Dim strComputer

Dim objWMIService

Dim colItems

strComputer = "."

Set objWMIService = GetObject("winmgmts:\\" & _

 strComputer & "\root\cimv2")

Set colItems = objWMIService.ExecQuery( _

 "Select * from Win32_OperatingSystem",,48)

For Each objItem in colItems

 WScript.Echo "BootDevice: " & objItem.BootDevice

 WScript.Echo "BuildNumber: " & objItem.BuildNumber

 WScript.Echo "BuildType: " & objItem.BuildType

 WScript.Echo "Caption: " & objItem.Caption

 WScript.Echo "CodeSet: " & objItem.CodeSet
```

Using the Query Results

Let's look at a real-world use for WMI, and walk through the process of building the script. Suppose you want to modify a remote computer's network configuration so that all network adapters have DHCP enabled. Actually, you'll probably want to check multiple machines at once, so you'll need the script to read computer names from a text file that you'll create, using one computer name per line within the file. If the script finds that DHCP is already enabled, you want it to tell you so.

NOTE

A slightly more real-world task might be to modify the configuration only for a specific network adapter, like the one named Local Area Network, in each machine. That requires working with WMI associator classes, which I'll cover in the next chapter.

The first part I like to handle is the WMI bit. I've found the Win32_NetworkAdapterConfiguration class, which has an EnableDHCP method that should do the job. I used the PrimalScript WMI Query Wizard to generate a template script for the class, and then trimmed it down to look like this.

```
On Error Resume Next

Dim strComputer

Dim objWMIService

Dim colItems



strComputer = "."

Set objWMIService = GetObject("winmgmts:\\" & _

 strComputer & "\root\cimv2")

Set colItems = objWMIService.ExecQuery( _

 "Select * from Win32_NetworkAdapterConfiguration",,48)

For Each objItem in colItems

 WScript.Echo "DHCPEnabled: " & objItem.DHCPEnabled

 WScript.Echo "Caption: " & objItem.Caption

Next
```

I need to have the script run through a text file, so I'll add the appropriate code. I showed you how to work with files and folders in Chapter 12.

```
Dim strComputer

Dim objWMIService

Dim colItems
```

Dim objFSO, objTS

Alternative Methods

As I mentioned earlier in this chapter, you're likely to run across other ways of performing WMI queries. For example, here's a short script that returns some information about a remote machine named Server1.

```
Set System = GetObject("winmgmts:{impersonationLevel=" & _
 "impersonate}!//server1/root/cimv2:" & _
 "Win32_ComputerSystem=""SERVER1""")

WScript.Echo System.Caption
WScript.Echo System.PrimaryOwnerName
WScript.Echo System.Domain
WScript.Echo System.SystemType
```

This doesn't follow the template-style query I've been using so far; in fact, it doesn't even use WQL. However, this example is functionally the same as the following one.

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

strComputer = "server1"
Set objWMIService = GetObject("winmgmts:\\" & _
 strComputer & "\root\cimv2")
Set colItems = objWMIService.ExecQuery( _
 "Select * from Win32_ComputerSystem WHERE " & _
 "Name = 'SERVER1'",,48)
For Each objItem in colItems
 WScript.Echo "Caption: " & objItem.Caption
 WScript.Echo "Domain: " & objItem.Domain
 WScript.Echo "PrimaryOwnerName: " & objItem.PrimaryOwnerName
 WScript.Echo "SystemType: " & objItem.SystemType
Next
```

There is practically no difference between the two. The first example uses GetObject to connect directly to a

Review

You've seen several examples of how to query basic WMI information in this chapter. I showed you how to look for the proper WMI classes, write more complex WQL queries, and test your queries. I also showed you how to start with a wizard-created template script and modify it to suit your needs, even if those needs involve changing something or performing an action, rather than simply displaying or retrieving WMI information. I've shown you examples of how WMI can be queried in different ways that will help you work with the many different examples you'll find on the Web and in other publications.

All of this will help you work with most of the simpler WMI classes. Some classes, however, represent more complex bodies of information, and have to be handled a bit differently. I'll cover those in the next chapter.

COMING UP

Working with complex information like shares, file and folder security, and user accounts requires you to build on the basic WMI skills you've learned so far. In the next chapter, I'll introduce you to associator classes and references in WMI, and explain how to work with more complex information and interrelated WMI classes.

Chapter 19. Querying Complex WMI Information

IN THIS CHAPTER

You've seen how to perform basic WMI queries and work with the results; in this chapter, I'll explain how more complex classes interrelate with one another and how you can work with them inside of your scripts. You'll use these techniques for working with a number of security- and configuration-related tasks, and they'll truly make WMI a powerful administrative tool.

In the previous chapter, I briefly described how some WMI classes have complex interrelationships with other classes, and promised to show you—in this chapter—how to deal with the information contained in those relationships. I even mentioned specific WQL keywords, including REFERENCES OF and ASSOCIATORS OF—that are used to query these complex classes. Now it's time to dive in and put them to work.

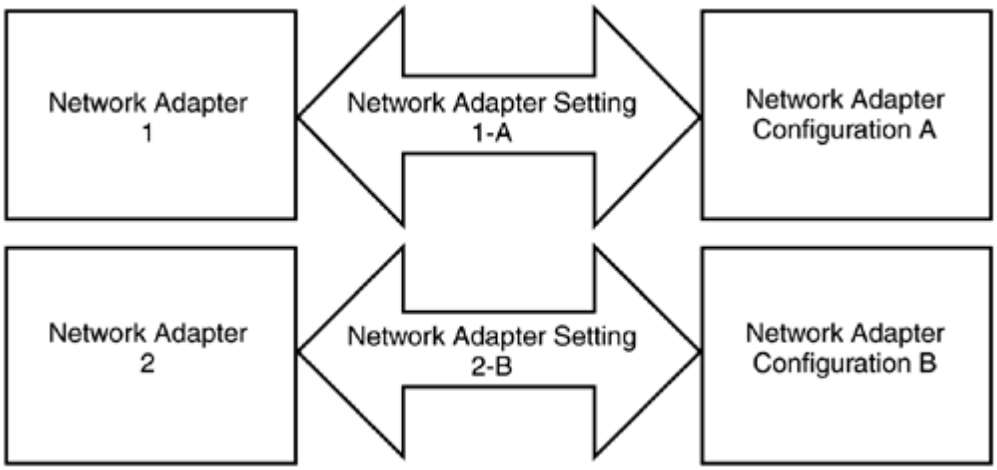
Understanding WMI Relationships

Probably the best way to understand the more complex WMI classes is with an example. Take Win32_NetworkAdapter. This class represents a physical network adapter inside a computer, whether it's an Ethernet adapter, an IEEE 1394 (FireWire) adapter, or whatever. If you examine the class' properties in the WMI documentation, you'll see that it only includes properties that deal with the physical hardware, such as its MAC address, whether it supports media sense (which tells Windows that a cable is unplugged), its maximum speed, and so forth.

WMI also defines a class named Win32_NetworkAdapterConfiguration, which includes the software aspects of a network adapter, including its IP address, IPX settings, and so forth. In theory, a single hardware adapter can have multiple possible configurations, which is why these properties are split into two classes. In fact, it's theoretically possible for one configuration to be shared by two different physical adapters. WMI needs some way to relate the two classes to one another, and that way is called an associator class. In this case, the associator class is Win32_NetworkAdapterSetting, which associates a network adapter and its configuration settings.

An examination of Win32_NetworkAdapterSetting's documentation reveals that it has only two properties: Win32_NetworkAdapter and Win32_ NetworkAdapterConfiguration. In other words, the two properties refer back to the associated classes. The associator, then, represents a single combination of adapter and configuration, as illustrated in Figure 19.1.

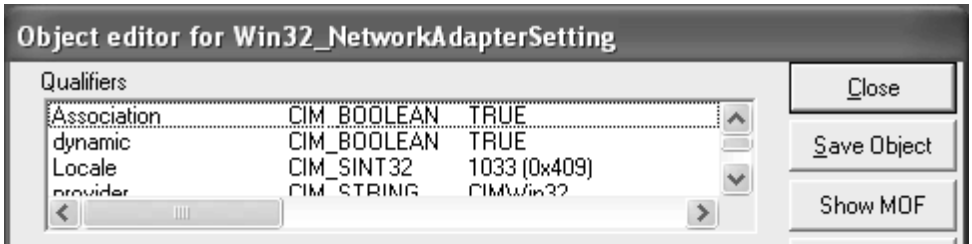
Figure 19.1. Associating two classes



You'll notice that neither the Scriptomatic nor the PrimalScript WMI Query Wizard includes Win32_NetworkAdapterSetting on their list of classes. That's because neither tool is designed to deal with associator classes, so you are on your own for working with them. You can get a bit of help from the Wbemtest tool, however. (I showed you how to use Wbemtest in Chapter 17.)

First, run Wbemtest and connect to your local computer's root\cimv2 namespace. Then, click Open Class and open the Win32_ NetworkAdapterSetting class. You should see a dialog box like the one in Figure 19.2. Of particular interest are the two main properties: Element and Setting. According to the WMI documentation, this class' Element represents a Win32_NetworkAdapter, and the Setting represents an associated Win32_NetworkAdapterConfiguration.

Figure 19.2. Examining the Win32_NetworkAdapterSetting class



Associating WMI Instances

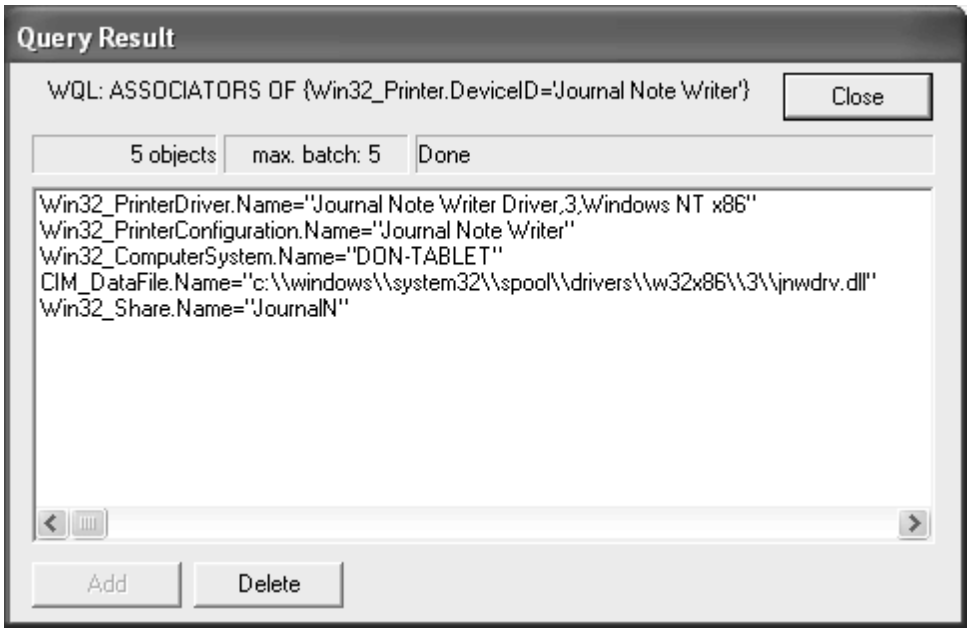
Hopefully, my previous two examples make it easier for you to understand WMI associations. However, they're bad examples for truly working with associated classes. Why? Because you aren't ever going to begin knowing which instance of the associator class you want; you're going to begin with one of the associated classes instead. Using the preceding technique, suppose you want to find the shares for a particular printer. You'd have to

- Get the correct Win32_Printer class first to get its DeviceID.
- Query Win32_PrinterShares for all instances where the Antecedent property references the DeviceID you're looking for.
- Take the results of that query and retrieve all referenced instances of Win32_Share.

ASSOCIATORS OF

The aforementioned technique is an awkward way to get the information, and that's why WQL offers the ASSOCIATORS OF command. Suppose you have a printer with a device ID of "LaserJet 5." You've created three or four shares of the printer, each with different permissions or whatever. You want to use WMI to retrieve the name of each share, and list the maximum concurrent number of users allowed to use each share. You could write a WQL query like this: ASSOCIATORS OF {Win32_Printer.DeviceID = "LaserJet 5"}. Note that ASSOCIATORS OF replaces the SELECT, property list, FROM, and class name elements of a more traditional WQL query. Also note that the class must be listed in curly braces {} not parentheses. That messes me up every time. Figure 19.3 shows the results of this query in Wbemtest (assuming you have a printer named LaserJet 5, that is; for this example, I used a different printer name).

Figure 19.3. Results of the ASSOCIATORS OF query



It turns out there are several associated classes:

- Win32_PrinterDriver
- Win32_PrinterConfiguration

Writing the Query

You've seen the whole associated class thing in action, but I want to start fresh with a new example and walk you through the entire query- and script-creation process. In the last chapter, I showed you how to set all the network adapters on a computer to use DHCP. In this chapter, I want to be more specific, and only modify the properties of a specific network adapter within the computer. More specifically, I want to

- Read a list of computer names from a text file
- Connect to WMI on each computer and locate the network adapter named "Local Area Connection"
- Ensure that each configuration for that adapter is set to use DHCP

It seems like the following query should do what I want.

```
ASSOCIATORS OF

{Win32_NetworkAdapter.NetConnectionID="Local Area Connection"}

WHERE

RESULTCLASS = Win32_NetworkAdapterConfiguration
```

That should pull all Win32_NetworkAdapter instances where the NetConnectionID is "Local Area Connection," and then retrieve the associated Win32_NetworkAdapterConfiguration instances.

Testing the Query

Wbemtest is the place to test my new query. Unfortunately, executing it yields an error: "Invalid object path." Uh-oh.

I'm guessing the problem is that Win32_NetworkAdapter and Win32_NetworkAdapterConfiguration are associated through Win32_Net-workAdapterSetting, which uses Win32_NetworkAdapter.DeviceID and Win32_NetworkAdapterConfiguration.Index to perform the association. In other words, the associator class has no clue about Win32_Network-Adapter.NetConnectionID.

Just to confirm that, I'll retest the query using this.

```
ASSOCIATORS OF {Win32_NetworkAdapter.DeviceID="1"} WHERE

RESULTCLASS = Win32_NetworkAdapterConfiguration
```

Sure enough, this query returns the expected instance of Win32_NetworkAdapterConfiguration. Here's what I'm going to have to do.

- Read a list of computer names from a text file.
- Connect to WMI on each computer and locate the network adapter named "Local Area Connection."
- Get the DeviceID from the Win32_NetworkAdapter instances returned.
- For each instance, query the associated Win32_NetworkAdapterConfiguration instances.
- For each of those instances, ensure that each configuration for that adapter is set to use DHCP.

I just need to code these actions into a script.

Writing the Query in VBScript

Now it's time to incorporate the query into a script. This time, I'll start with the shell of the script, which will read the computer names from the text file.

```
Dim oFSO, oTS, sComputer


Set oFSO = CreateObject("Scripting.FileSystemObject")

Set oTS = oFSO.OpenTextFile("c:\input.txt")

Do Until oTS.AtEndOfStream


 sComputer = oTS.ReadLine


Loop
```

That is easy enough. Now, for each, I need to retrieve a specified instance of Win32_NetworkAdapter. The caption I'm looking for—"Local Area Connection"—is stored in a property named NetConnectionID.

TIP

How did I know which property to use? Simple: Wbemtest. I clicked EnumInstances and typed Win32_NetworkAdapter as the superclass name. Then, I double-clicked on the first instance that was returned to display its properties. I scrolled down, looking for "Local Area Connection" in the values column, and I found it in a property named NetConnectionID. If I hadn't found "Local Area Connection" at all, I would have tried the next instance in the list, and kept browsing until I found it.

Actually, I don't want to retrieve the Win32_NetworkAdapter instance at all. Instead, I need to retrieve all associated Win32_NetworkAdapterConfiguration instances. However, as I discovered earlier, I need to retrieve the DeviceID on my own, based on a simpler WQL query. Here's the modified script.

```
Dim oFSO, oTS, sComputer
```

Dim oWMI, oConfigs, oConfig, oAdapters, oAdapter

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

Set oTS = oFSO.OpenTextFile("c:\input.txt")


Do Until oTS.AtEndOfStream


 sComputer = oTS.ReadLine
```

Another Example

This business of using associator classes is complicated, so I'm including an additional example of how they work. For this example, let's say you want to list all of the shared folders on a particular file server, along with the physical file path that each share represents. For each of those physical folders (or directories), you want to enable NTFS file compression. Here's what you need to do.

- Connect to WMI on a specified server.
- Retrieve a list of Win32_Share class instances that represent file shares (as opposed to printer or other shares)
- For each instance, retrieve the physical folder as a Win32_Directory class.
- For each physical folder, use the Compress method.

➤➤ Compressing All Shared Folders

Listing 19.2 shows the entire script you'll need to use.

Listing 19.2. CompressAll.vbs**. This script compresses all shared folders on a specified file server.**

```
'get server name

strComputer = InputBox("Server name?")


'connect to WMI

Set objWMIService = GetObject("winmgmts:" & _

 "\\" & strComputer & "\root\cimv2")


'retrieve the list of shares

Set colShares = objWMIService.ExecQuery _

 ("SELECT * FROM Win32_Share WHERE " & _

 "Type = 0")


'for each share returned...

For Each objShare In colShares


 'retrieve the associated folders

 Set colFolders = objWMIService.ExecQuery _
```

Review

In this chapter, I've shown you how different WMI classes can be related to one another through associator classes. I've also introduced you to the WQL ASSOCIATORS OF query, which allows you to query those relationships. You've learned how to use Wbemtest to test your queries, incorporate your queries into a script, and then utilize the query results to perform administrative tasks.

By now, you should have a solid understanding of how WMI works from within a script, and how you can use it to both query and modify configuration settings within your computers. You should feel comfortable working with the simpler queries that Scriptomatic or the PrimalScript WMI Query Wizard can generate for you, and you should be comfortable writing more complex queries that utilize WMI associations and class relationships. As always, of course, the toughest part about WMI is figuring out which classes to query, but hopefully by now you're becoming comfortable with the WMI class reference in the MSDN Library, and you're able to browse through the class list and select the appropriate classes.

COMING UP

You've seen how to work with all kinds of WMI information, so it's time to pull everything together into a complete script. In the next chapter, I'll provide you with two complete WMI and ADSI sample scripts, so you can see how these technologies are used in real-world situations. Then, in Part IV, I'll show you how to leverage everything you've learned so far to create Web-based administrative scripts that make a great addition to your administrative utility belt.

Chapter 20. Putting It All Together: Your First WMI/ADSI Script

IN THIS CHAPTER

It's time to leverage what you've learned about ADSI and WMI scripting. In this chapter, I'll walk you through the entire design and creation process for a new script. In addition to demonstrating a useful new purpose for WMI and ADSI, this chapter will help strengthen your script design skills.

By now, you should have a good idea of what WMI and ADSI can do for you. In this chapter, I'll walk you through the complete design process for an entirely new script. This time, I'll use both WMI and ADSI in the same script. The script's job will be to check in on every computer in an Active Directory or NT domain and query some information about its operating systems. I want the script to output this information to a text file on a file server. The information I want to collect includes operating system version, service pack level, number of processors in the machine, maximum physical memory in the machine, and so forth. This is a useful way to quickly inventory a network and see what machines might need to be upgraded before deploying a new application, or to see what machines don't have the latest service pack applied.

Designing the Script

My script is a reasonably complex undertaking, so it helps to break it down into manageable tasks. I need the script to do three things:

1. Query a list of computers from the domain.
2. Query information from each computer.
3. Write information out to a text file.

The last bit is probably the easiest. I can use the FileSystemObject to open a text file, write information to it, and then close the text file. Something like the following would work.

```
Dim oFSO, oFile

Set oFSO = CreateObject("Scripting.FileSystemObject")

Set oFile = oFSO.CreateTextFile("output.txt")

oFile.Write "Information"

oFile.Close
```

For more information on using the FileSystemObject, refer to Chapter 12.

Querying a list of computers from the domain shouldn't be too hard, either. If I want the script to work with both NT and Active Directory domains, I need to use the WinNT ADSI provider, because only that provider works with both domains. I can query all of the objects in the domain, and then use an If…Then construct to work with only the computer objects. Code such as the following should do the trick.

```
Dim oDomain

Set oDomain = GetObject("WinNT://" & sDomain)

Dim oObject, sComputerName, sDetails

For Each oObject In oDomain


 'is this object a computer?

 If oObject.Class = "Computer" Then


  'yes - do something with it


 End If

Next
```

Writing Functions and Subroutines

The one bit of functionality that seems to be standalone is the code generated by the wizard, which will do my WMI querying for me. I may need to use that code in another script someday, and I'll definitely be using it over and over in the script I'm writing now, so it makes sense to write it as a function.

I want the function to accept a computer name, query that computer for specific operating system information, and then compile all that information into a neatly formatted string. The function should return the string to the main script, which can then write it to a file or whatever.

Adapting the wizard's code isn't too difficult. Listing 20.2 shows my new GetOSIno() function. Note that this isn't intended to be run as a standalone script; as a function, it must be called by another script, which must provide the name of the computer to connect to as the function's input parameter.

Listing 20.2. GetOSInfo.vbs. **This function queries a computer's operating system information and returns the results in a string.**

```
Function GetOSInfo(sComputer)


      'declare variables

      Dim objWMIService

      Dim colItems

      Dim strOutput


      'get WMI service

      Set objWMIService = GetObject("winmgmts:\\" & _

       strComputer & "\root\cimv2")


      'get item collection

      Set colItems = objWMIService.ExecQuery( _

       "Select * from Win32_OperatingSystem",,48)


      'init output string

      sOutput = String(70,"-")

      sOutput = sOutput & sComputer


      'append info to output string

      For Each objItem in colItems

            strOutput = strOutput & "BuildNumber: " & _

             objItem.BuildNumber & vbCrLf
```

Writing the Main Script

The function was probably the toughest part to write; with that out of the way, I can adapt my prototype code to create the main script, shown in Listing 20.3.

Listing 20.3. MainScript.vbs. **Queries the domain, creates the output file, and calls the custom function I already wrote.**

```
Dim sDomain

sDomain = InputBox("Enter domain to inventory")


'connect to domain and retrieve

'a list of member objects

Dim oDomain

Set oDomain = GetObject("WinNT://" & sDomain)


'get the filesystemobject

Dim oFSO

Set oFSO = CreateObject("Scripting.FileSystemObject")


'open an output file

Dim oOutput

Set oOutput = oFSO.CreateTextFile("\\server1\public\output.txt")


'run through the objects

Dim oObject, sComputerName, sDetails

For Each oObject In oDomain


 'is this object a computer?

 If oObject.Class = "Computer" Then


  'yes - get computer name

  sComputerName = oObject.Name


  'get OS info

  sDetails = GetOSInfo(sComputerName)
```

Testing the Script

If you jumped ahead and already tried to execute the final script, you realize that it's flawed. If you haven't, go ahead and give it a whirl now. Take a few minutes to see if you can track down the problem. There are actually three errors, and here are some hints.

- One is a simple typo.
- One is a sort of logic error, where something isn't being used properly for the situation.
- The last one is a typo, and could have been avoided if I had followed my own advice from earlier in the book.

Can you find them all? The first one is an easy mistake: I simply forgot a closing parentheses.

```
'connect to domain and retrieve

'a list of member objects

Dim oDomain

Set oDomain = GetObject("WinNT://" & sDomain
```

The correct code should be Set oDomain = GetObject("WinNT://" & sDomain). The next one's a bit trickier.

```
'open an output file

Dim oOutput

oOutput = oFSO.CreateTextFile("\\server1\public\output.txt")
```

Can you see it? I'm using oOutput to represent an object, but I forgot to use the Set keyword when making the assignment. VBScript requires Set whenever you're assigning an object to a variable. The corrected code looks like this.

```
'open an output file

Dim oOutput

Set oOutput = oFSO.CreateTextFile("\\server1\public\output.txt")
```

The last error is tricky, too. It's in the GetOSInfo() function.

```
Function GetOSInfo(sComputer)


      'declare variables
```

Review

Pulling together ADSI and WMI into a single script offers some powerful functionality. More importantly, though, the example in this chapter should make you feel more comfortable with the sometimes-daunting task of creating a script from scratch. Just break down the tasks that need to be completed, and then develop some prototype code for each task. Use wizards, examples from the Web, or samples from this book to help create prototype code. After all, there's no sense reinventing the wheel when there's a large library of samples on the Web and in this book to work with!

With your task list and prototype out of the way, you can start assembling the script. Write functions and subs to perform repetitive tasks, or tasks that you may want to reuse in future scripts. Write the main script, and then start testing. With this methodology in mind, most scripts can be whipped together quickly!

COMING UP

Web pages offer an exciting way to create your own centrally located, easily accessible administrative tools. In the next chapter, I'll introduce you to Active Server Pages, and in the following chapters, I'll show you how to easily and quickly apply your scripting skills to create great administrative Web pages.

Part IV: Creating Administrative Web Pages

Chapter 21. Active Server Pages Crash Course

IN THIS CHAPTER

You'll never bother or you'll use it all the time: Active Server Pages, Microsoft's scripted Web pages, provides a powerful way to create administrative Web pages and user self-service pages. Although making you a full ASP programmer is beyond the scope of this book, there's plenty of power available to you through the VBScript that you've already learned.

Web pages are still the hottest technology around, especially with the recent hype about Web Services and other new Web-based technologies. Web pages can have a place in administration, too. I'm not talking about creating full-fledged Web applications (although you certainly could do so by using VBScript), but you can create some great administrative utilities that are based upon Web pages. Consider the following useful, easy-to-create solutions:

- A Web page that allows Help Desk technicians to quickly check the status of a particular user account
- A Web page that retrieves inventory information about client computers on the network
- A Web page that allows users to reset their own passwords by providing some piece of personally identifiable information

The keys to the kingdom are Active Server Pages, or ASP, Microsoft's scripted Web development environment.

About ASP

ASP was first introduced with Internet Information Server 3.0 (IIS 3.0), and exists in subsequent versions of IIS. It's important that you make a distinction between ASP and ASP.NET. ASP is Microsoft's original server-side scripting model, whereas ASP.NET is a completely new, .NET Framework-based technology that has nothing to do with scripting at all. IIS 5.0 and 6.0 support ASP.NET, and both of them support running both ASP and ASP.NET applications within the same Web site. Although ASP.NET is a powerful, high performance way to write Web applications, it's a bit outside the scope of administrative scripting. For instance, it doesn't use VBScript; so-called "Classic" ASP does. So, for the purposes of this book, I'll focus on the older ASP technology.

ASP is not a programming language, although that is a popular misconception. ASP is simply a specialized object model and a specialized host for VBScript. The ASP object model allows you to access information that users type into HTML input forms, and allows you to write output to HTML pages. In fact, ASP isn't particularly high-tech or fancy (although it was pretty innovative when it was introduced).

To get started with ASP, consider a basic HTML Web page.

```
<HTML>

<BODY>

<FORM ACTION="display.asp" METHOD="POST">

Computer name: <INPUT TYPE="TEXT" NAME="COMPUTERNAME"><BR>

<INPUT TYPE="SUBMIT">

</FORM>

</BODY>

</HTML>
```

Type this HTML code into Notepad and save the file as Display.asp. The file needs to be located within an IIS folder, such as C:\Inetpub\Wwwroot. If you've got the file in the correct place, you should be able to select Run from the Start menu, type http://localhost/display.asp, and see a small Web page that says "Computer name:" and has a text box and a Submit button.

Is this ASP? Not really. This is just HTML—the Web page doesn't actually do anything if you click the Submit button. You could create a page like this with any HTML editor, such as Microsoft FrontPage. In fact, I usually use FrontPage to work with ASP. You can also use higher-end script editors like PrimalScript, because they understand both ASP tags and HTML code.

Getting Ready for Web Scripting

For this chapter and the three that follow, I'm going to assume your computer is set up to run ASP pages. You'll need a Windows 2000, Windows Server 2003, or Windows XP computer with IIS installed. You'll also need to ensure that ASP is enabled; this is the default for Windows 2000 and Windows XP, but you'll have to explicitly enable ASP in Windows Server 2003.

You should also know where the IIS root folder is. Generally, that's in C:\Inetpub\Wwwroot; you can check the properties of the Default Web Site in the Internet Services Manager console to see where the root folder is located on your computer. I'll have you place all of the sample Web pages in that root folder.

VBScript in ASP

Where's the scripting in ASP? You have to add it. Essentially, the same ASP page has a mix of both VBScript code and HTML code. The VBScript code has to be added between two special tags: <% and %>. These tags tell IIS where the script code begins and ends, and allows it to distinguish between VBScript and HTML. Consider the following revision to our running example.

```
<HTML>

<BODY>

<%

Response.Write("It is now " & Now)

%>

<FORM ACTION="display.asp" METHOD="POST">

Computer name: <INPUT TYPE="TEXT" NAME="COMPUTERNAME"><BR>

<INPUT TYPE="SUBMIT">

</FORM>

</BODY>

</HTML>
```

Believe it or not, that's a full ASP page. The key is in the VBScript line Response.Write("It is now " & Now), which is a line of VBScript that uses ASP's built-in Response object to output the current date and time to the Web page. Try running that—remember, save the page in an IIS folder and access the page by using your Web browser and the HTTP protocol—and you'll see what it looks like.

Here's how it works: When you request the file from the Web server, IIS locates the appropriate file on the hard drive. It loads the file into memory, and because the file has an .ASP filename extension, IIS hands it off to Asp.dll, which is an ISAPI plug-in that handles ASP pages. Asp.dll scans each line of the file. The first two lines obviously aren't code, because Asp.dll hasn't seen a <% tag; therefore, those lines are passed straight through to IIS, and IIS transmits them to your Web browser.

On the third line, ASP realizes that some script code is beginning. Asp.dll reads the fourth line, and executes it. The results of the code execution are passed to IIS and transmitted to your Web browser. On the fifth line of the file, Asp.dll stops looking for code and continues passing everything straight through to IIS.

TIP

If you're feeling lazy, you don't have to type out Response.Write; you can simply use the equal sign. <% = "Hello" %> is functionally the same as <% Response.Write "Hello" %>.

If you've loaded this page into your Web browser, right-click the page and select View Source from the context menu (assuming you're using Internet Explorer, of course). You should see something like this.

```
<HTML>

<BODY>
```

The Response Object

You've already seen a quick example of the Response object and its most important method: Write. The Response object is used to output information to the client Web browser, including text, cookies, HTML headers, and more. In an administrative script, you're not likely to use much more than Response.Write, and it's one of the two additional commands that I mentioned you'd have to learn.

Writing Output

Response.Write simply outputs text to the Web page. Think of it as a sort of MsgBox statement, or more accurately a WScript.Echo for Web pages. As you've seen, you can include functions and literal text, and Response.Write simply outputs whatever you tell it to.

Here's another example of how Response.Write works. Save this page as Response.asp in your computer's Web root folder, and access it via http://localhost/response.asp.

```
Response.Write Now & "<br><br>"

Response.Write "Hello!" & "<br><br>"

Response.Write "All done!"
```

Use View Source in your Web browser to see the final HTML that was transmitted to the browser.

TIP

The
 tags I use in this example are HTML tags for a line break. They're similar to using vbCrLf in a regular script, and tell the Web browser to insert a carriage return and linefeed. Use two of them in a row, as I've done, to create blank lines. Using vbCrLf doesn't work in an ASP script, because Web browsers tend to ignore incoming carriage returns and linefeeds when they display HTML.

Saving Cookies

The Response object also allows you to save cookies to client computers. A cookie is a small collection of data, normally smaller than 1024 bytes in size. A cookie is a collection of crumbs (seriously), with each crumb representing one piece of data. So, if you want to save the user's name, that would be one crumb in the cookie, the user's last logon date might be another. In administrative scripts, cookies tend to have limited use. One potential use is in a Web-based wizard, such as a new user creation wizard. In that application, you might use a cookie to keep track of the settings the user enters on each page of the wizard; on the last page, you could then collect all that data together to create the new user account.

Response provides access to cookies through the Response.Cookies collection. It's simple to use; here's an example of setting two crumbs.

```
Response.Cookies("UserName") = "JohnD"

Response.Cookies("AcctExpires") = 0
```

The trick with cookies is that they have to be passed in the HTTP headers, not in the main HTML code. That means you have to set the cookies before any HTML is output to the browser. The following example works fine.

The Request Object

If the Response object provides a means of producing output, you might suspect that the Request object allows you to work with user input—and you'd be right. Primarily, the Request object lets you access cookies, and lets you access information entered into HTML forms.

HTML Forms

Whenever you add an HTML form field to a Web page, you give it a name. For example, <INPUT TYPE="TEXT" NAME="ComputerName"> creates a text box named "ComputerName." When that form is submitted back to the Web browser (when the user clicks a Submit button), whatever the user entered is paired with the form field name to make it more readily accessible to you.

ASP exposes form fields through its Forms collection. You can use these just like read-only variables, as shown in this snippet.

```
The computer name you entered was:

<% Response.Write Request.Forms("ComputerName") %>
```

Any fields not filled in by the user will be blank, and equal to "". You can use Request.Forms in logical comparisons, to produce additional output (as in the preceding example), and so forth. You'll see plenty of the Request object in the full-length sample later in this chapter.

TIP

As a shortcut, you can omit Forms when reading forms input. For example, instead of typing Request.Forms("ComputerName"), you could simply type Request("ComputerName") and get the same results. I'll use that technique in most of my examples to save space.

Cookies

Just as Response.Cookies allows you to write cookies to client Web browsers, Request.Cookies allows you to read them back out again. For example, if you've saved a crumb named "UserName," Request.Cookies("UserName") will read the crumb. Here's an example.

```
<HTML><BODY>

You requested account creation for user name

<% Response.Write Request.Cookies("UserName") %>.<BR>
```

Request.Cookies can be used anywhere in a script as it doesn't attempt to modify the HTTP headers being sent to the client.

A Sample ASP Script

As an example, I'll show you a simple ASP script that accepts the name of a computer, and then uses Windows Management Instrumentation (WMI) to display some key information about that computer. You could expand this easily into a real-time inventory check page, usable by help desk technicians, perhaps.

Before this script will work, however, you need to make some changes to the computer running it. I'm going to list the changes needed; I'll describe the need for these changes in Chapter 23.

1. Open the Registry Editor.
2. Browse to HKEY_LOCAL_MACHINE\Software\Microsoft\WBEM\Scripting. This key should already exist.
3. Create a new DWORD value named "Enable for ASP" and set the value to 1. If the value already exists, make sure it's set to 1.
4. If you're putting this Web page in the Default Web Site, use Internet Services Manager to open the properties of that Web site. If you're using another Web site, open that site's properties.
5. On the Directory Security tab, disable anonymous access and ensure that Windows Integrated Authentication is selected.

When you attempt to access the Web page (or any other page in that Web site), you may be prompted to log on. Make sure you're doing so with a user account that's an administrator, or the script may not run properly. ASP and WMI security are complex; I'll cover them in more detail in Chapter 23 to explain what's going on.

➤➤ Displaying Information in a Web Page

The script in Listing 21.1 uses WMI to display information about a selected computer.

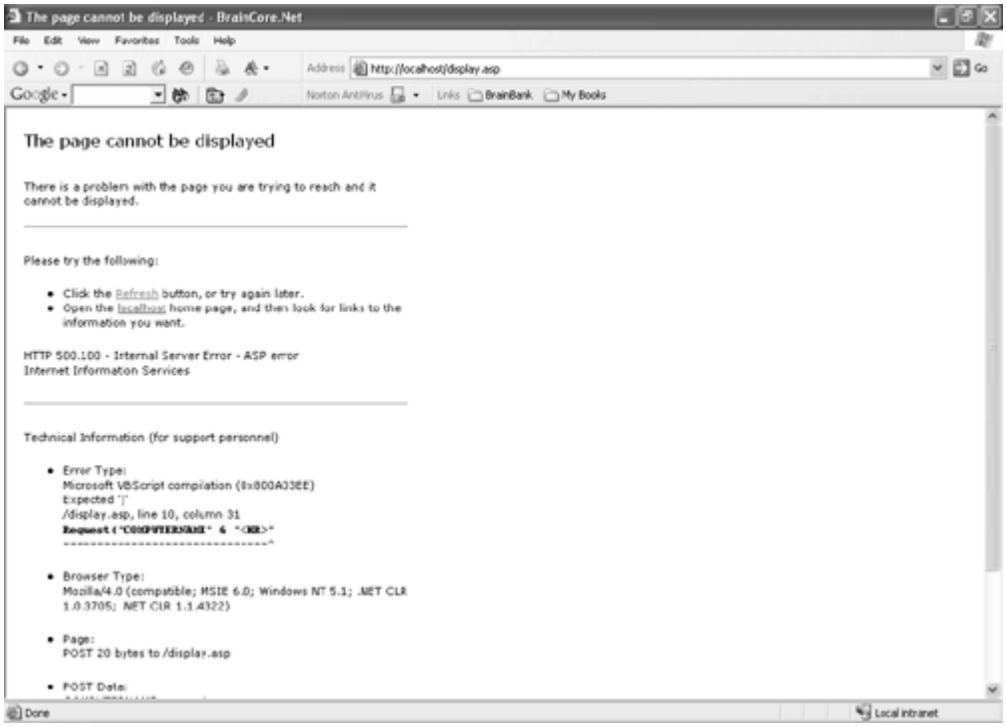
Listing 21.1. Display.asp. **Uses WMI to display information about a computer.**

```
<%


If Request("COMPUTERNAME") <> "" Then


 Set oSystem = GetObject("winmgmts:{impersonationLevel=" & _

  "impersonate}!//" & Request("COMPUTERNAME") & _

  "/root/cimv2:Win32_ComputerSystem=" & Chr(34) & _

  Request("COMPUTERNAME") & Chr(34))


 Response.Write "System information for " & _
```

Testing ASP Scripts

Testing ASP scripts can be a bit of a pain. Typically, an error results in a browser window like the one shown in Figure 21.1.

Figure 21.1. ASP errors display in the browser window



The technical information in the error message breaks down as follows:

```
HTTP 500.100 - Internal Server Error - ASP error

Internet Information Services


Technical Information (for support personnel)


Error Type:

Microsoft VBScript compilation (0x800A03EE)

Expected ')'

/display.asp, line 10, column 31
```

Request("COMPUTERNAME" & "<HR>"

The important information is under "Error Type," and in this case, the error is that VBScript was expecting to see a closing parentheses and it didn't. The offending line of code is helpfully displayed. It looks like I forgot to include a ")" after Request("COMPUTERNAME", and that's causing the error. It's helpful to have an editor like FrontPage or PrimalScript that displays line and column numbers. You can jump straight to the line in question, but beware of the column number: In this example, ASP is giving me column 31, which is actually at the end of the line, but the error is at column 23. ASP isn't always good about pinpointing errors.

Review

ASP can be a great way to extend administrative script functionality to other administrators in your enterprise. With it, you can create Web utilities that don't necessarily require administrative privileges, and you can closely control the functionality provided by your Web tools. ASP also makes it easier to provide access: Rather than distributing scripts via e-mail or a file server, they can be centrally located and easily accessible from a single intranet Web server.

The capability of ASP to perform actions that ordinary users can't perform can also be useful. For example, I work with one company that allows department heads to submit new user account requests online. The ASP page waits for administrator approval, and then creates the new user account automatically. Administrators don't need to type in any additional information, and the approval process can be handled by a junior administrator or assistant who might not have Domain Admin privileges. ASP is a powerful way to leverage your growing scripting skills, especially in large organizations.

COMING UP

With the ASP basics under your belt, you're ready to build a basic static Web page and add some VBScript to it, and I'll show you how in the next chapter. After that, I'll explain some of the finer points of ASP security, and then show you how to pull everything together in a Web page designed for Help Desk technicians.

Chapter 22. Adding Administrative Script to a Web Page

IN THIS CHAPTER

Making administrative Web pages isn't hard, but it can be tough to figure out where to start. I'll walk you through the process step-by-step, and you'll be coding your own pages before you know it.

Administrative Web pages essentially have two elements: HTML and VBScript. The two work together, and often intermingle, within a single Web page, and that can make it tough to figure out where to start when you're creating a Web page. It's easy once you get used to it, though, and in this chapter, I'll show you an example. I'll create a Web page that allows users to pull event log entries from any computer, provided they have the proper credentials to do so. The script will also allow filtering by event ID and source, making it a useful tool for combing through large event log files. While this is certainly a task you can accomplish with the Computer Management console, having this available as a Web page makes it readily accessible to network administrators, and makes it a bit more single-purpose, which can help junior administrators focus on the task at hand.

The Basic Web Page

I always start by using FrontPage or a similar Web page layout tool to design the page's look and feel, before I add any script. Now, I'm no Web page designer, so my Web pages aren't usually super attractive, but they get the job done. Administrative scripts usually don't have to be as pretty as they do functional!

For this example, I know I'm going to need to start with some kind of HTML input form that will allow the user to specify a computer, an event log, an event ID, and other criteria. That type of form is easy enough to create in FrontPage, and here's the HTML code.

```
<FORM ACTION="event.asp" METHOD="POST">
 <Table cellpadding=2 cellspacing=2 border=0>
  <TR>
   <TD>
    <INPUT type="text" name="ComputerName"
     value="">
   </TD>
   <TD>Computer:</TD>
  </TR>
  <TR>
   <TD>
    <SELECT name="LogName">
     <OPTION value="application">Application</OPTION>
     <OPTION value="system">System</OPTION>
     <OPTION value="security">Security</OPTION>
     </SELECT>
    </TD>
    <TD>Log</TD>
   </TR>
   <TR>
    <TD><INPUT type="text" name="Source"></TD>
    <TD>Event Source</TD>
   </TR>
   <TR>
    <TD>
     <SELECT name="Type">
      <OPTION value="">All</option>
```

Adding Functions and Subroutines

From some reading, I know that two pieces of data may cause me problems. Each event log entry has a date and time associated with it, and WMI returns those with kind of a funny format. I need to take what WMI gives me and reformat it into normal-looking dates and times. I may as well create some functions to handle the reformatting.

```
Function CDateWMI(cim_DateTime)
 'declare variables
 Dim sDateTime, iYear, iMonth, iDay

 'convert the date to a string
 sDateTime = CStr(cim_DateTime)

 'get the year, month, and day
 iYear = CInt(Mid(sDateTime, 1, 4))
 iMonth = CInt(Mid(sDateTime, 5, 2))
 iDay = CInt(Mid(sDateTime, 7, 2))

 'reformat into a normal date
 CDateWMI = CDate(Join(Array(iMonth, iDay, iYear), "/"))
End Function

Function CTimeWMI(cim_DateTime)
 'declare variables
 Dim sDateTime, iHours, iMinutes, iSeconds

 'convert the time into a string
 sDateTime = CStr(cim_DateTime)

 'get the hours, minutes, and seconds
 iHours = CInt(Mid(sDateTime, 9, 2))
 iMinutes = CInt(Mid(sDateTime, 11, 2))
 iSeconds = CInt(Mid(sDateTime, 13, 2))

 'reformat into a normal time
```

Adding Inline Script

Now it's time to start pulling everything together. In the previous chapter, I mentioned how it can save time and maintenance effort to keep a form and the handling page in the same file. I showed you an example of how to use an If…Then construct to tell if the form is being submitted or not.

In this page, I want to only display the HTML page if the form isn't being submitted. If it is being submitted, I want to take the submitted information and display the appropriate event log entries.

ASP allows you to include conditional static HTML in a Web page. It's actually harder to describe than to do it, so here's an example.

```
<%


'Was the form submitted?

If Request.Form("SUBMIT") = "" Then


%>

<FORM ACTION="event.asp" METHOD="POST">

 <Table cellpadding=2 cellspacing=2 border=0>

  <TR>

   <TD>

    <INPUT type="text" name="ComputerName"

     value="<% Response.Write CompName%>">

   </TD>

   <TD>Computer:</TD>

  </TR>

  <TR>

   <TD>

    <SELECT name="LogName">

     <OPTION value="application">Application</OPTION>

     <OPTION value="system">System</OPTION>

     <OPTION value="security">Security</OPTION>

     </SELECT>

    </TD>

    <TD>Log</TD>

   </TR>

   <TR>
```

The Result

The final script is a combination of the original static HTML, the functions I created, and the inline script code I created. Merging everything results in the working event log Web viewer.

➤➤ Event Log Viewer

Listing 22.1 shows the final event log viewer, in its entirety.

Listing 22.1. Event.asp. **Displays local and remote event logs in a Web browser.**

```
<%


'Was the form submitted?

If Request.Form("SUBMIT") = "" Then


 'No - set up the form

 'Start by getting the local computer name

 Set oNet =CreateObject("WScript.Network")

 sCompName=oNet.Computername

 Response.Write "Computer: " & compname & "<BR>"

 Set oNet = Nothing


 'Now display the HTML form

%>

<FORM ACTION="event.asp" METHOD="POST">

 <Table cellpadding=2 cellspacing=2 border=0>

  <TR>

   <TD>

    <INPUT type="text" name="ComputerName"

     value="<% Response.Write CompName%>">

   </TD>

   <TD>Computer:</TD>

  </TR>

  <TR>

   <TD>

    <SELECT name="LogName">
```

Review

I hope that the example in this chapter has shown you how easy it can be to create useful administrative Web pages. Start by designing the Web page itself in FrontPage or another editor. Don't worry about producing a beautiful page, just get the elements in place that you need. Then, add functions and subroutines as necessary to support the script you'll write. Finally, add any inline code that you need. Keep in mind that ASP allows static HTML to be conditional if placed within an If…Then construct.

COMING UP

ASP, IIS, WMI, ADSI, and the other technologies used in scripting can conspire to create real security vulnerabilities—or security hassles, depending on how you look at it. In the next chapter, I'll explain the pros and cons of ASP scripting security and show you how to configure your machines to provide functionality with minimal vulnerabilities.

Chapter 23. Web Page Security Overview

IN THIS CHAPTER

IIS' built-in security model can make administrative Web pages a bit more difficult to write. I'll explain how IIS handles security, and how your scripts and Web servers need to be configured in order to enable scripting functionality.

Administrative Web pages must comply with IIS' security features. Those features differ a bit from version to version; in this chapter, I'm assuming that you're using Internet Information Server (IIS) 4.0 (Windows NT), IIS 5.0 (Windows 2000), IIS 5.1 (Windows XP), or IIS 6.0 (Windows Server 2003). IIS 6.0 differs the most from a security standpoint, so I'll spend extra time discussing those differences.

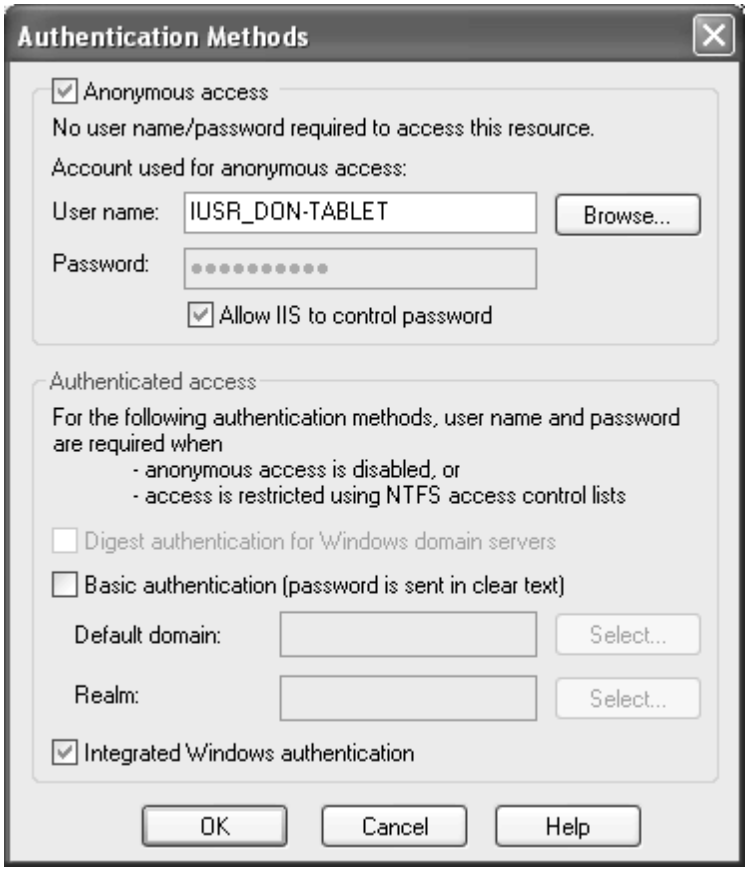
Bear in mind that IIS' entire security makeup is designed to prevent you from doing what administrative Web pages do, particularly working with ADSI and WMI. That's OK, though, because it's configurable, and I'll show you how to make it work.

The ASP Security Context

ASP pages run under a specific security context, and the identity of that context depends on a few factors. First, bear in mind that the IIS service itself runs under the LocalSystem account. As the name implies, LocalSystem has almost complete control over all local resources, and practically no authority to access network resources. However, although IIS runs under the LocalSystem account, actions performed by Web pages (including ASP pages) do not have LocalSystem privileges. That's because IIS uses a system of impersonations to grant security privileges to Web pages.

The first impersonation IIS does is the Anonymous user account. That's the IUSR_machinename user account that's created when IIS is installed. It's a local account, not a domain account, and it isn't a member of the machine's Users group—it's a member of the Guests group. That membership severely restricts the functionality that the account has, and pretty much limits it to reading Web pages. IIS performs this impersonation whenever anonymous access is enabled for a Web site, as shown in Figure 23.1.

Figure 23.1. Enabling anonymous access for a Web site



NOTE

Windows Server 2003 doesn't use the IUSR_machinename account by default. Instead, it uses a special new built-in user account named Network Service. Regardless, many administrators commonly change IIS to use a more restricted user account, which may affect what your scripts can do.

Note that you can change the user account used to grant anonymous access. I don't recommend ever using a more powerful account, though, as you enable completely unauthenticated users to perform some powerful operations. For example, the Anonymous user account doesn't have permission to run WMI scripts or queries; you could configure IIS to use the local Administrator account for anonymous access instead. That would get your WMI scripts running, but it would also allow complete strangers to have full control over your Web server. Not a good idea!

Remember, IIS uses the Anonymous user account whenever anonymous access is allowed, and when there are no special NTFS permissions on the Web page a user requests. If you want to force IIS to use another form of

Prohibited Behaviors

To protect your computers and your network, the Anonymous IIS user account is not a member of the local Users group; it's a member of the Guests user group. By default, a large number of useful scripting features and operating system functionality aren't available to members of the Guests group:

- WMI.
- Most ADSI functions; only general LDAP queries are allowed for most directory services. Active Directory doesn't usually allow anonymous LDAP queries by default.
- Most operations that use the FileSystemObject (FSO).
- Most operations using Windows Script Host (WSH) objects, including the Network and Shell objects.
- Almost all network operations, including using any components that require network access.

The Anonymous user account is a local account on the Web server, and as such has no privileges or capabilities across the network or within a domain. Essentially, you can use intrinsic and custom VBScript functions and features, the intrinsic ASP objects (including Request and Response), and that's about it.

NOTE

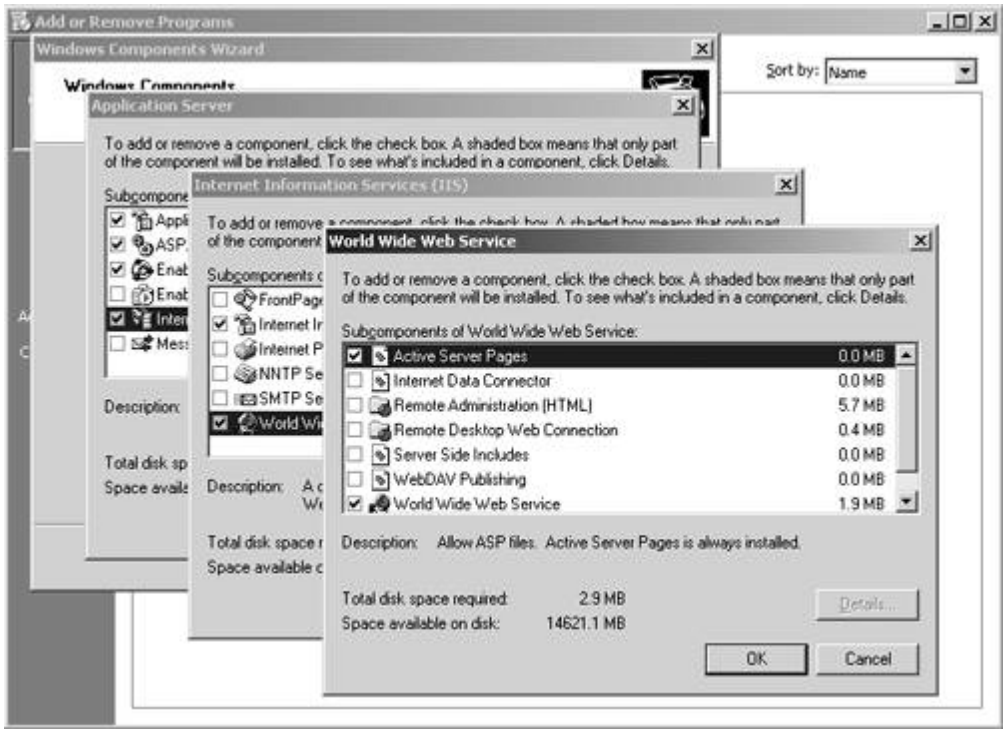
IIS itself doesn't impose many restrictions on functionality; it's the user context being used to execute ASP pages that can be restricted. The Anonymous account can't do much, whereas an administrator account can do anything. Always keep that simple rule in mind as you write administrative Web pages.

IIS 4.0, 5.0, and 5.1 versus IIS 6.0

IIS 6.0 is the first time Microsoft has tried to create a more secure IIS "out of the box," as opposed to a more functional IIS. That's led to a lot of misconceptions, but keep in mind that IIS 6.0 isn't necessarily more secure than earlier versions—it's just that most of the functionality that includes security vulnerabilities is disabled by default on a new IIS 6.0 installation. Upgrades from previous versions are not necessarily more secure out of the box, as they inherit the settings from the old version being upgraded.

One of the first things you need to know about IIS 6.0 is that ASP is not installed by default. In other words, if you just select the Application Server component in Add/Remove Windows Components, IIS 6.0 will be installed without the ASP ISAPI filter, meaning ASP pages won't be executed. You must go into the Details of the component installation to enable ASP, as shown in Figure 23.3.

Figure 23.3. Installing ASP support with IIS 6.0



IIS 6.0 also adds another authentication choice: Microsoft .NET Passport authentication. This authentication type requires special setup and developer support, as well as a paid license to the .NET Passport system. It's not something you'll typically use in an administrative script.

Much of IIS 6.0's new security features aren't actually part of IIS 6.0 at all; they're part of the .NET Framework, including ASP.NET, which is bundled with Windows Server 2003. However, "classic" ASP doesn't utilize any of the Framework's security features, so I won't cover them here.

NOTE

If you're an experienced IIS administrator and you'd like to learn more about what's new and changed in IIS 6.0, check out Microsoft® IIS 6.0 Delta Guide by Martin C. Brown, published by Sams Publishing.

IIS 6.0 does have an interesting new concept called application pools, which have their own security features. The application pools play into IIS 6.0's new architecture. First, each application pool has its own memory space, meaning a crash in one pool won't affect applications running in other pools. From a security standpoint, each pool has its own security context. By default this is a network service account, but you can also configure it to be a local system account or any specific user account you prefer, as shown in Figure 23.4.

Figure 23.4. Configuring application pool identity

NTFS and IIS Security

IIS is fully integrated with the Windows operating system, which means it respects and obeys Windows' security constraints. In particular, IIS respects the permissions assigned to files and folders that reside on NTFS-formatted volumes.

TIP

Because FAT (and its variants, like FAT32) doesn't provide any file-level security, I don't recommend using it to store Web site content. Always store Web site content—especially administrative Web pages—on an NTFS volume.

The bottom line is this: IIS won't allow any user to access a Web page unless that user has at least Read permissions to the Web page file. You can use this to your advantage when you create administrative Web pages. Simply edit the security properties of your Web pages, and ensure that only authorized users have Read access. Remove the Everyone group, for example, and add the Domain Admins group. Doing so forces IIS to authenticate any user requesting the page—even if anonymous access is enabled—and prevents unauthorized access.

Writing Secure ASP Code

Obviously, I recommend using NTFS permissions to secure your administrative Web pages. Grant permissions only to administrators that are allowed to use the pages, and completely remove the Everyone group from the permissions. Also, ensure that the Guest group doesn't have permissions. After you've done that, IIS won't allow anyone who isn't authorized to even see the pages, let alone try to execute them.

You can use a couple of other techniques to improve the security of your administrative Web pages. First, authenticate your users. For example, if you've written a script that only members of the Help Desk group should be able to execute, use a function like the one in Listing 23.2.

Listing 23.2. CheckMembership.vbs**. This function ensures a user is a member of a particular group.**

```
Function AllowUser(sDomain, sAuthorizedGroup)


 Dim sUserADPath, bLoggedOn

 sUserADPath = "WinNT://" & sDomain & _

  "/" & Request.ServerVariables("LOGON_USER")


 'change backslashes to slashes for ADSI

 sUserADPath = Replace(sUserADPath, "\", "/")


 'get the group

 Set oGroup = GetObject("WinNT://DON-TABLET/" & _

  sAuthorizedGroup & ",group")


 'is the user a member?

 For Each oMember in oGroup.Members

  If LCase(oMember.ADsPath) = LCase(sUserADPath) Then

   bLoggedOn = True

   Exit for

  End If

 Next


 AllowUser = bLoggedOn

End Function
```

This function accepts a domain name and a group name, and returns True if the current user is a member of that group. Use it as follows

Review

Understanding ASP and IIS security isn't difficult, and you need to master the basic concepts in this chapter before you can start effectively using administrative Web pages. The most important lesson, as in any security discussion, is to trust no one. Always check input values and parameters to make sure they're what you expected, and always validate user identity before performing any tasks. ASP and VBScript provide some functionality to make identity validation and data scrubbing easy, and IIS itself provides a number of features to help secure and protect your administrative Web pages.

COMING UP

In the next chapter, I'll help you put it all together with two complete, systematic examples of designing and writing administrative Web pages. I'll build on the examples in Chapters 13 and 20, where you practiced your script design skills, and show you how to start with a basic static HTML page and create a functional, useful Web-based administrative tool.

Chapter 24. Putting It All Together: Your First Administrative Web Pages

IN THIS CHAPTER

It's time to take everything you've learned and create a complete, administrative Web page from scratch. The two step-by-step examples in this chapter will include the design process, as well as some cool ADSI code that queries domain information.

It's time to take everything you've learned about scripting and Web pages and merge them. In this chapter, I'm going to walk you through two complete examples. The first example will be a simple one: creating a Web page that allows junior administrators to quickly and easily check the status of a user account in a domain. The second example will be a bit more complicated, and will allow an administrator to automatically create new IIS virtual directories. That example will show you how to check a user's identity from within a script—a useful trick in any administrative Web page.

Most importantly, these examples will walk you through the complete Web page design process, so that you'll feel more comfortable embarking on your own projects in the future.

Checking User Account Status

The business requirements for this example aren't complex. I want to have a simple Web page that accepts a user ID, and then queries a predetermined NT or Active Directory domain to see if that user account is locked out or disabled. I want the results to be displayed on the bottom of the page, so that an administrator can quickly enter another user ID to check its status.

Designing the Page

I use Microsoft FrontPage to produce most of my Web page designs. It's easy, and with FrontPage 11, you don't get a lot of extraneous fancy HTML formatting, so it's easier to see what's going on.

Listing 24.1 shows the simple HTML form I cooked up in FrontPage.

Listing 24.1. Userprops.htm**. This is a static HTML page that will be adapted to ASP later.**

```
<html>

<head><title>Check User Account Status</title></head>

<body>

<p align="center"><b>Check User Account Status</b><i><br>

This page will work only for domain administrators

or account operators.</i></p>

<form method="POST" action="userprops.asp">

 <p align="center">User ID:<br>

 <input type="text" name="UserID" size="20"></p>

 <p align="center"><input type="submit"

 value="Submit" name="Submit"></p>

 <hr noshade color="#000000">

</form>

<p align="center"><b>Status for<br>

<u>xx<br>

</u>yy</b></p>

</body>

</html>
```

If you open the page in FrontPage or even a Web browser, you'll notice that I've included some placeholder information. The "xx" and "yy" at the bottom of the page are where I want my script to insert the user ID and its account status. I find that including placeholder information like this helps me figure out exactly where in the HTML code my script needs to insert stuff. It also lets me apply formatting—such as boldfacing and underlining—by using FrontPage's excellent formatting tools, so that I don't have to worry about what the underlying HTML codes are.

TIP

Administering IIS

Suppose you want to make it easy for junior administrators to create new virtual directories under IIS. You need a tool that will create the virtual directory, and automatically assign the designated Web developer as the owner of the new directory. This would be a great tool in a Web development shop, and can save a lot of mouse clicking. Suppose you also want to make sure that only members of a special Web Operators group on the Web server can use the page.

NOTE

This code is adapted in part from the IIS Admin Web site included with IIS 4.0 and 5.0. That site and its source are great examples of how to use the IIS administration object, which is utilized within this script.

Designing the Page

As usual, I start with FrontPage to make the basic HTML. In this case, it's pretty simple, and it's shown in Listing 24.4.

Listing 24.4. IIS.htm. **Static HTML for the IIS administration page.**

```
<HTML>

<BODY>

<CENTER>

 <FORM ACTION="iis.asp" METHOD="POST">

  <INPUT type="text" name="VirtualDir">

  <SELECT size=1 name="Developer">

   <OPTION VALUE="test">TEST</OPTION>

  </SELECT>

  <INPUT type="checkbox" name="AllowScript">

  <INPUT type="submit" value="Submit" name="Submit">

 </FORM>

</CENTER>

</BODY>

</HTML>
```

Not much to it. The only placeholder information is the TEST option in the drop-down list box; I'll want to populate the list with the members of the Web Developers group.

For a script this complex, I also want to lay out the tasks that I need to accomplish.

- Create an IIS virtual directory.
-

Review

The example in this chapter should help you see how a typical administrative Web page is designed, created, tested, and used. Although this was a relatively simple example, it offers powerful functionality. You could expand this example to work in a multiple-domain environment, perhaps offering a drop-down list of available domains, or even a drop-down list of users (although that might take a while to populate in a large domain). The point is that Web pages offer an exciting, easy-to-use alternative interface for junior administrators, can be made as secure as other types of administrative utilities, and are easy to create by using the same techniques that you use for other scripts.

COMING UP

Enough of Web pages—it's time to move on to more advanced general scripting techniques. In the next chapter, I'll introduce you to modular programming, including script packaging and Windows Script Components. In following chapters, you'll see how to protect scripts by using encryption, and you'll learn more about scripting and security.

Part V: Advanced Scripting Techniques

Chapter 25. Modular Script Programming

IN THIS CHAPTER

Want to do more scripting in less time? Then you need to get modular, which is what this chapter is all about. You'll learn to create code in an easy-to-reuse fashion, enabling future scripts to leverage the work you've done in the past.

Throughout this book, I advocate the use of functions and subs to encapsulate useful script routines. This type of encapsulation makes it easy to cut and paste functions and subs into future scripts, allowing you to easily reuse script code that may have taken you a while to write and debug. Cutting and pasting is great, but it has some fundamental flaws. For example, if you decide to improve a particular function, you have to improve every copy of it that you've ever made, one copy at a time. Fortunately, VBScript provides a more efficient way to reuse the functions and subs you write: Windows Script Components (WSC).

Introduction to Windows Script Components

To properly introduce WSC, I need to dive a bit deeper into developer-speak than I'm accustomed to doing; so bear with me. First, you should realize that you're already using programming objects in your scripts. Specifically, you're using objects—or components—written to Microsoft's Component Object Model, or COM. I briefly touched on COM in Chapter 5, but here's a quick refresher on what it does for you.

When you create, or instantiate, a COM class in a script, you do so by using the CreateObject statement. For example, CreateObject("Scripting.FileSystemObject") creates a new FileSystemObject. When VBScript executes that command, it asks COM to load "Scripting.FileSystemObject" into memory and make it available to VBScript. COM looks up the class "Scripting.FileSystemObject" in the registry. You can open the registry yourself, using Regedit or another editor, and search for "Scripting.FileSystemObject." You'll find that it has a globally unique identifier (GUID) of "{0D43FE01-F093-11CF-8940-00A0C9054228}" and that it's in-process server (InprocServer32) is C:\Windows\System32\scrrun.dll, which is the Microsoft Scripting Runtime DLL. COM loads that DLL into memory when you ask for a FileSystemObject.

All COM components must have an in-process server. When you create a new WSC, you're essentially creating a script that pretends to be a COM component. That pretense is helped by Scrobj.dll, which is the WSC in-process server. You can create instances of WSCs within scripts, and when you do so, COM loads Scrobj.dll, which in turn loads the actual WSC script and executes it. So a WSC is a regular VBScript masquerading as a DLL! In fact, any programming language that uses DLLs—including Visual Basic, Delphi, VBScript, C++, and more—can use a WSC, because WSCs meet all of the requirements for regular COM components.

Okay, that's enough developer-ese for one chapter. It's time to start looking at how you create these things.

In- and Out-of-Process

What, exactly, is the difference between in-process and out-of-process? In IIS 5.0 (and earlier), each Web site on a server runs in a separate instance of Inetinfo.exe. Each Inetinfo.exe has its own memory space, and any COM objects that an ASP page instantiates generally run inside that memory space. It's easy for programmers, but if a COM object causes a problem, it can take down the entire Web site. Almost all objects you'll use in ASP run in process; out-of-process components run in their own memory space, and must implement special techniques to allow communications between their own space and Inetinfo.exe's.

IIS 6.0 introduces a slightly new model called worker process isolation. Without going into the gory detail of IIS 6.0's architecture, it works something like this: Only one copy of Inetinfo.exe runs on the server. It spawns several subordinate worker processes, each with its own memory space, to execute Web sites. In-process servers run in these worker process spaces, effectively protecting different Web sites from a crash in any one site.

Scripting and XML

WSCs are regular text files, but they require a special XML formatting to contain the script. The XML helps describe what the WSC does, and how it is activated and used. Probably the easiest way to see how it works is to see an example, so take a look at Listing 25.1. This is a sample WSC that performs several Windows Management Instrumentation (WMI) functions. I adapted the script from one first provided on www.wshscripting.com, an unfortunately now-defunct Web site that offered scripting examples.

Listing 25.1. WMIFunctions.wsc**. Example WSC.**

```
<?xml version="1.0"?>

<package>

 <comment>

 WMI Management Library

 </comment>

 <component id="WMILIB">

  <?component error="true" debug="true" ?>

  <registration progid="WMILIB.WSC"

  classid="{61E6E0DC-4554-4D12-A9F4-D8E70DBCF318}"

  description="WMI Library" remotable="no" version="1.00">

  </registration>

  <public>

   <method name="Shutdown">

    <parameter name="Host"/>

   </method>

   <method name="Reboot">

    <parameter name="Host"/>

   </method>

   <method name="StartProcess">

    <parameter name="Host"/>

    <parameter name="CommandLine"/>

    <parameter name="StartDirectory"/>

   </method>

   <method name="Processes">

    <parameter name="Host"/>

   </method>

   <method name="EndProcess">
```

Review

Windows Script Components, or WSCs, are special scripts that can be executed like COM components. They make it easy to package, redistribute, and reuse scripts and routines that have taken you a long time to perfect, thus making your scripting efforts faster and more efficient. WSCs are written much like normal scripts, but have a special XML layout that allows them to be executed by Scrobj.dll.

COMING UP

With the fundamentals of Windows Script Components under your belt, you're ready to start reusing code in your scripts. In the next chapter, I'll show you how to use components in a real script, and I'll show you how you can integrate other modular programming techniques.

Chapter 26. Using Script Components

IN THIS CHAPTER

You've seen how to create script components; in this chapter, I'll show you how to utilize them in your own scripts. There's a ton of script components on the Web that serve valuable functions, and you don't need to know much about how they work to use them in your own scripts.

In the previous chapter, I showed you how you can create your own script components. They're a great way to modularize and encapsulate your code, allowing you to easily reuse it in other scripts. However, for administrators, the real value of script components is the ease with which they allow you to use other people's work in your own scripts. In this chapter, I'll guide you through the process of locating, registering, and utilizing the script components that have been written by hundreds of other script programmers and made available on the Web.

Obtaining the Component

Hop on Google and find the component you need. I usually try to come up with some keywords that match the task I'm trying to do, and then throw "script component" in quotes, along for the ride. That usually delivers some good results in a page or two of hits.

For this walk-through, I've selected a component that allows me to display a more complex dialog box than VBScript provides. You can get the component's source code from http://cwashington.netreach.net/depo/view.asp?Index=409&ScriptType=component; you need to download it onto your local computer to use it. The component is credited to Micheal Harris.

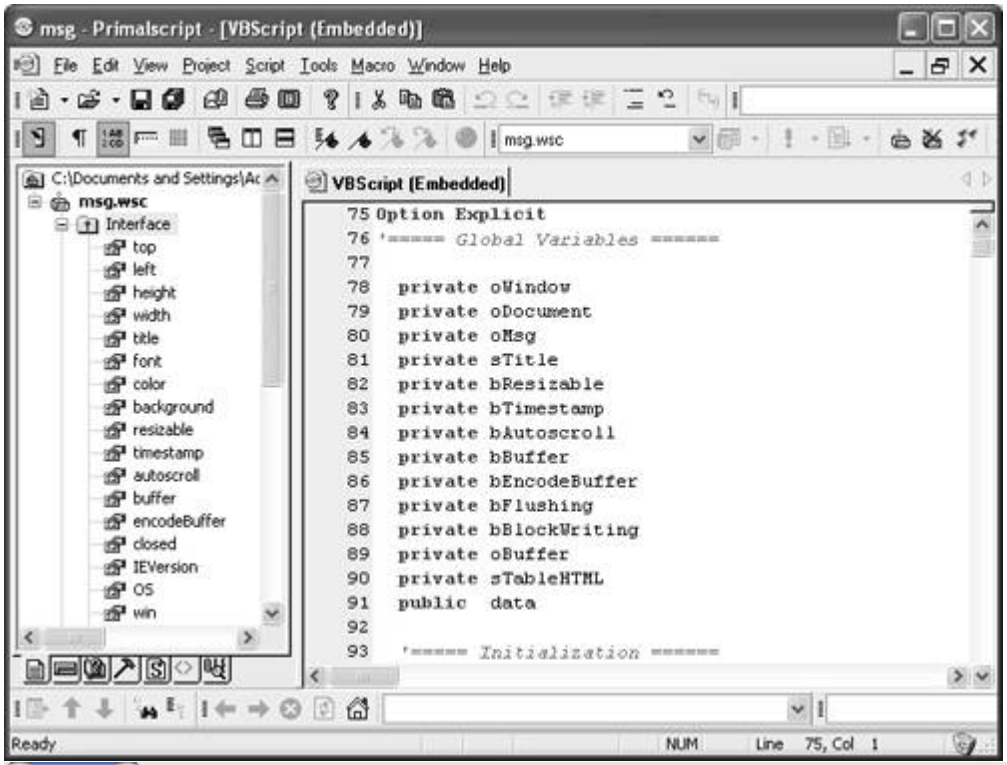
Reviewing the Component

Given the nasty things people can do with script viruses, I never run a component without looking at it first. Opening MSG.wsc in PrimalScript, or simply reviewing the source code online, reveals a relatively simple script. It's in VBScript, so I can more or less follow what it's doing. I don't see any FileSystemObject calls to delete everything on the computer, any code that looks like it's going to send an e-mail to everyone in my address book, and so forth.

Reviewing the component also allows me to see what methods and properties it offers. In this case, I can get that information from the Web site where the component is found, but that might not be the case for every component you run across. Looking through this component in PrimalScript, I can see its interfaces. As shown in Figure 26.1, PrimalScript lists each method and property in its left-hand tree. You can click on any one to jump straight to the code that implements each. This script appears to implement the following:

- A Show method
- A Write() method, which accepts a line of text as a parameter
- A Clear method
- Several properties such as Top and Left, which appear to set the size of the dialog box
- A number of other useful-looking properties and methods

Figure 26.1. Reviewing a script component's interfaces



Using the Component

Next, I need to register the component, which is easy enough. Right-click it in Explorer and select Register from the Context menu. Now I'm ready to use the component in a script.

➤➤ Showing a Message Box

Listing 26.1 shows the script I wrote that uses the MSG.wsc component.

Listing 26.1. ShowMessageBox.vbs**. Displaying a progress bar.**

```
'show the message box

     dim oMsgBox

     set oMsgBox = createobject("msg.wsc")

     oMsgBox.show




     oMsgBox.write "Custom text goes here"

     oMsgBox.write "Close this box to continue"


oMsgBox.complete
```

This script should work fine, provided you've downloaded and registered the component.

➤➤ Showing a Message Box—Explained

This script starts simply enough, by declaring a variable and creating an object reference to the MSG.wsc component that I downloaded and registered.

```
'show the message box

dim oMsgBox

set oMsgBox = createobject("msg.wsc")
```

Next, I simply show the message box and use its WriteLine method to display some text.

```
oMsgBox.show




oMsgBox.write "Custom text goes here"

oMsgBox.write "Close this box to continue"
```

Review

Script components are easy to create, and even easier to use. By using script components, you can leverage your own past work as well as the work of others to create more powerful, professional, and flexible scripts—all much faster than if you had to do everything on your own from scratch. Also, keep in mind that I'll provide links to additional components as I find them (and as you suggest them) on this book's companion Web site, www.adminscripting.com.

COMING UP

If you're interested in protecting your scripts and creating a safer scripting environment, the next two chapters are for you. I'll cover script encoding and scripting security issues, along with suggestions for configuring your environment for safer scripting.

Chapter 27. Encoded Scripts

IN THIS CHAPTER

Looking to protect the intellectual property or the integrity of your scripts? Encoding is the way to go, and WSH supports a complete script encoding technology. You'll learn how to use it, and how it can benefit your administrative scripts.

Do you ever worry about your users seeing your script source code and somehow learning more than they should? Or, perhaps you just want to ensure that your scripts aren't modified and run by someone who shouldn't be doing so. Encrypted or encoded scripts offer a solution to these problems, and Microsoft offers a Script Encoder tool to use with your administrative scripts. The Encoder can take a script and turn it into something like this.

```
//**Start

Encode**#@~^QwIAAA==@#@&0;mDkWP7nDb0zZKD.n1YAMGhk+Dvb`@#@&P,kW`UC7kLlDG

Dcl22gl:n~{'~Jtr1DGkW6YP&xDnD+OPA62sKD+ME#@#@&P,~~k6PvxC\rLmYGDcCwa.n.k

kWUbx[+X66Pcr*cJ#,@*{~!*P~P,P~. YEMU`DDE bIP,P,+s/n@#@&P~P,~PM+O;Mx`WC^

/n#pN6EU1YbWx,o Obaw.WaDrCD+nmL+v#@#@&~P7lMPdY.q,'~J_CN,Y4rkP4nnPCx,C1Y

;mV, +(PkrY ~~l,wCL PmKhwmYk(snPSkDt~JI@#@&P~\m.PkY.+,'PE8MWA/ .kPGDt D

PDtmUPri@#@&,P-CMP/D.&,'Pr\rmMWkWWY~(YnDnY,2a2^WDn.,*!,Ep@#@&,P7lD,/D.

c,'~JSW;s9Ptm-+,4+ U~VKl9+[REI,Pr0,c\ DrWHZW.. mOAMGS/nM`*#@#@&P,

~P9W^Es+UOchDbO+v/YMq~_,/DDfPQ~kY.c*IP,+sd @#@&~~,P[W1;s+UDRSDkD+vdYMF

~_,/O.yP_,dYM&P3~dYMc*iNz&R @*^#~@
```

Although it looks like gibberish, it still runs perfectly. You can be assured that nobody will change the script, because changing a single character of the encoded script will render it useless.

Installing the Script Encoder

Microsoft's Script Encoder can be downloaded from the Scripting Web site at http://msdn.microsoft.com/scripting. Just click the Download link and look for the Script Encoder. You can find complete documentation online at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/seconscriptencoderoverview.asp.

The Script Encoder is a command-line tool, and is designed to run against an already-written and debugged script. After you encode the script, you cannot change it; if you do need to make changes, you have to work with the original unencoded version and then re-encode the changed script.

Encoded versus Encrypted

The Script Encoder looks like a form of encryption. In a way, in fact, it is a form of encryption: Clear-test script code is run through a mathematical algorithm and the result is illegible (at least to humans). The Windows Script Host understands how to decode the script, though, allowing it to retrieve the original script code and execute it. Therefore, the Encoder can be said to use a form of encryption.

However, the Encoder isn't designed to foil all attempts at accessing your source code. All scripts are encoded using the same algorithm, so that any copy of the Windows Script Host can decode and execute the script. That means it isn't impossible—or even necessarily difficult—for a clever person to figure out the encoding algorithm used and create his own decoder.

You can rely on the Encoder to stop casual access to your source code, and to stop casual users from attempting to modify your scripts. However, you cannot rely on the Encoder to provide absolute protection for your scripts.

Writing Encoded Scripts

You write encoded scripts the same way you would almost any script, at least to start. Listing 27.1 shows an example.

Listing 27.1. ResetPW.vbs. **An unencoded administrative script written in VBScript.**

```
'get user account

varUserID = inputbox ("Reset password for what user ID?")


'bind to domain user

set objUser = GetObject("WinNT://MyDomain/" & varUserID & _

 ",user")


'make up a random password

varPass = DatePart( "y", Date() )

varPass = varPass & left(varUserID, 2)


'set password

objUser.SetPassword varPass


'show password

WScript.echo "New password is " & varPass
```

You don't need to add anything special to the file; the Script Encoder recognizes the VBS filename extension and deals with the file appropriately. To encode the file, simply run SCRENC /f resetpw.vbs. The Encoder produces a file named ResetPW.vbe, which is an encoded VBScript file. Here's what it will look like.

```
#@~^pAEAAA==@#@&BL Y,E/ D,Cm1W;xD@#@&-

mDjknD&fP{~rxaED4G6~crIn/ OPalddSWD[~6W.PS4mY~!/ DP&fQE#@#@&@#@&E4rU9PY

K~NK:lbU~Ek+M@#@&/nO,W8L`d+MPx~V+Y68N+^YvEqkUgK=zztXGG:mkUzrP'~7lD`d+Mq

f,'~JBEk+.Jb@#@&@#@&BsC3 P;2,lP.C

      NG:,2m/dSWMN@#@&\m.nm/dP{P9CD+nm.YvPJHESPGlD+c#~b@#@&-

lMKlk/~x,\l.Km/dPL~^+WD`7lD`/ .qG~~ *@#@&@#@&B/ OPal/kAGD9@#@&W8Lid D

? Onm/dAKDN~-mDKlkd@#@&@#@&BktWSPaC/khGD9@#@&

      UmDb2Yc+m4G~Jg+SP2lddSW.N,r/,J~',\l.Km/d@#@&@#@&2HoAAA==^#~@
```

Running Encoded Scripts

Encoded scripts can normally be executed just like any other script, with a couple of caveats. First, if your scripts don't include <SCRIPT> tags, the filename extensions must be either VBE (for VBScript) or JSE (for Jscript). The different filename extension tells WSH that it needs to decode the script before executing it; if you change the filename extension to VBS (or JS), you receive a runtime error when executing the script.

When the Encoder goes to work on a file that does use <SCRIPT> tags, it changes the LANGUAGE attribute. <SCRIPT LANGUAGE="VBScript"> becomes <SCRIPT LANGUAGE="VBScript.Encode">, for example, giving WSH the cue it needs to decode the script before trying to execute it.

Review

Script encoding offers a way to protect the source code of your scripts from prying eyes, and a way to ensure that your scripts aren't modified. Encoding doesn't provide any kid of runtime security; in other words, by default, Windows Script Host will execute any encoded script it's asked to execute. In the next chapter, I'll show you how to lock down WSH so that only your authorized scripts execute in your environment.

COMING UP

Encoding is a great way to secure and protect your scripts. In the next chapter, I'll explain how scripting itself can be made safer and more secure. Even in this age of script-borne computer viruses, you can still allow administrative scripts to run in your environment. After that, Part VI of this book will present several more full-length example scripts to start you toward your own administrative scripts!

Chapter 28. Scripting Security

IN THIS CHAPTER

Chances are that you've had to completely disable scripts in your environment, thanks to the number of abusive scripts out there. Making scripting a safe part of your environment can be difficult, so in this chapter, I'll give you some pointers for doing so.

Scripting has two primary security issues associated with it. First, the Windows Script Host (WSH) is included with just about every version of Windows since Windows 98. Second, WSH associates itself with a number of filename extensions, making it very easy for users to click an e-mail file attachment and launch unauthorized scripts. The knee-jerk reaction of many administrators is to simply disable scripting altogether, which also removes a beneficial administrative tool from the environment. In this chapter, I'll focus on ways to address the two primary security issues associated with scripting, helping you to configure a safer scripting environment.

Why Scripting Can Be Dangerous

"Why can scripting be dangerous?" isn't a question many administrators have to ask. Something like 70% of all new viruses, according to some authorities, are script based; certainly some of the most devastating viruses, including Nimda, Melissa, and others, propagate at least partially through scripts sent via e-mail. Even internally produced scripts can be dangerous, as scripts can delete users, create files, and perform any number—in fact, an almost unlimited number—of tasks. There's little question about the damage scripts can do, making it vitally important that your environment be secured to allow only those authorized, tested scripts that you or your fellow administrators authorize.

Perhaps the most dangerous aspect of administrative scripting is the easy accessibility scripts have to the system. Users can launch scripts without even realizing that they're doing so; a large number of file extensions are registered to the Windows Script Host, and double-clicking any file with one of those extensions launches the script. In Windows XP, the default script extensions are

- JS for JScript files
- JSCRIPT for Jscript files
- JSE for Jscript encoded files
- VBE for VBScript encoded files
- VBS for VBScript files
- WSC for Windows Script Components
- WSF for Windows Script Files

Note that older computers may also register VB for VBScript files, SCR for script files, and other extensions; Windows XP cleaned up the filename extension list a bit. Don't forget, of course, static HTML files—with HTML or HTM filename extensions—which can contain embedded client-side script.

NOTE

Other types of scripts exist, such as the Visual Basic for Applications (VBA) embedded into Microsoft Office documents. However, I'm going to focus this discussion on scripts associated or executed by the Windows Script Host.

The goal of any security program should be to allow beneficial, authorized scripts to run, while preventing unauthorized scripts from running.

Security Improvements in Windows XP and Windows Server 2003

Windows XP and Windows Server 2003 introduce a new concept called software restriction policies. These policies, which are part of the computer's local security settings and can be configured centrally through Group Policy, define the software that may and may not run on a computer. By default, Windows defines two possible categories that software can fall into: disallowed, meaning the software won't run, and unrestricted, meaning the software will run without restriction. Unrestricted is the default system security level, meaning that by default all software is allowed to run without restriction.

Windows also defines rules, which help to categorize software into either the disallowed or unrestricted categories. By default, Windows comes with four rules, defining all system software—Windows itself, in other words—as unrestricted. This way, even if you set the default security level to disallowed, Windows will continue to be categorized as unrestricted.

You can define your own rules, as well.

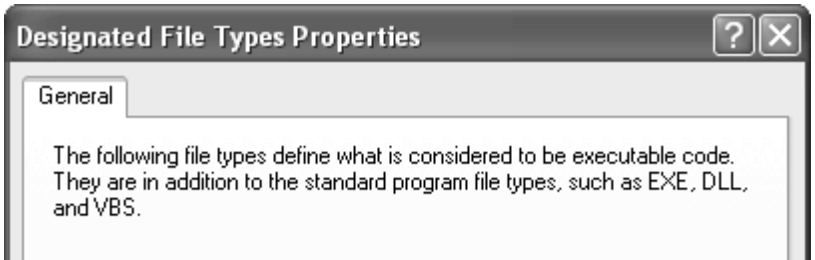
- Certificate rules identify software based on the digital certificate used to sign the software.
- Hash rules identify software based on a unique checksum, which is different for any given executable file.
- Path rules identify software based on its file path. You can also specify an entire folder, allowing all software in that folder to run or to be disallowed.
- Internet Zone rules identify software based on its Internet zone location.

Therefore, you create rules that allow Windows to identify software. The rules indicate if the identified software belongs to the unrestricted or disallowed categories. Software not specifically identified in a rule belongs to whichever category is set to be the system default.

Suppose, for example, that you set the system default level to disallowed. From then on, no software will run unless it is specifically identified in a rule and categorized as unrestricted. Although it takes a lot of configuration effort to make sure everything is listed as allowed, you can effectively prevent any unauthorized software—such as scripts—from running on your users' computers.

Software restriction policies also define a list of filename extensions that are considered by Windows to be executable, and the list includes (by default) many standard WSH scripting filename extensions. The DLL filename extension is notably absent from the list. That's because DLLs never execute by themselves; they must be called by another piece of software. By allowing DLLs to run unrestricted, you avoid much of the configuration hassle you might otherwise expect. For example, you can simply authorize Excel.exe to run, and not have to worry about the dozens of DLLs it uses, because they aren't restricted. The default filename extension list does not include JS, JSCRIPT, JSE, VBE, VBS, or WSF, and I heartily recommend that you add them. For example, Figure 28.1 shows that I've added VBS to the list of restricted filenames, forcing scripts to fall under software restriction policies.

Figure 28.1. Placing VBS files under software restriction policy control



Digitally Signing Scripts

A signed script includes a digital signature as a block comment within the file. You need to be using the WSH 5.6 or later XML format, because it contains a specific element for storing the certificate. Take Listing 28.1 as an example.

➤➤ Script Signer

This script signs another script for you. Just run it with the appropriate command-line parameters shown, or run it with no parameters to receive help on the correct usage.

Listing 28.1. Signer.vbs. **This script signs another one.**

```
<job>
 <runtime>
  <named name="file" helpstring="The script file to sign"
   required="true" type="string" />
  <named name="cert" helpstring="The certificate name"
   Required="true" type="string" />
  <named name="store" helpstring="The certificate store"
   Required="false" type="string" />
 </runtime>
 <script language="vbscript">

  Dim Signer, File, Cert, Store
  If Not WScript.Arguments.Named.Exists("cert") Or _
   Not WScript.Arguments.Named.Exists("file") Then

   WScript.Arguments.ShowUsage()
   WScript.Quit

  End If

  Set Signer = CreateObject("Scripting.Signer")
  File = WScript.Arguments.Named("file")
  Cert = WScript.Arguments.Named("cert")

  If WScript.Arguments.Named.Exists("store") Then
   Store = WScript.Arguments.Named("store")
```

Running Only Signed Scripts

If you don't want to mess around with software restriction policies, you can also rely on WSH's own built-in form of security policy. This policy allows you to specify that only signed scripts will be run; unsigned scripts won't be. This is probably the easiest and most effective way to prevent most unauthorized scripts.

To set the policy, open the registry key HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows Script Host\Settings\TrustPolicy. Set the value to 0 to run all scripts, 1 to prompt the user if the script is untrusted, and 2 to only run trusted scripts. What's a trusted script? Any script that has been digitally signed by a certificate that the user's computer is configured to trust. For example, if you purchase a certificate from VeriSign (which all Windows computers trust by default), and use that certificate to sign your scripts, they'll run. Unfortunately, a hacker could do the same thing—but you could easily investigate the source of the certificate, because it's a way to uniquely identify the signer.

Using this built-in trust policy allows you to run only signed scripts no matter what version of Windows your users have, provided you've deployed WSH 5.6 or later to all computers. Note that this technique, because it relies on WSH and not the operating system, works on all operating systems capable of running WSH. Many of the other techniques in this chapter—such as Software Restriction Policies—run only on Windows XP, Windows Server 2003, and later.

Ways to Implement Safe Scripting

Although Software Restriction Policies offer a promising way to control what runs on your users' computers, it's only available on XP and 2003, and does require some pretty significant planning before you can roll it out. Are there any alternatives to safely scripting? Absolutely.

The Filename Extension Game

One of the easiest ways is to configure your users' computers to no longer associate VBS, SCR, WSF, and other filename extensions with the WScript.exe executable. Removing these file associations prevents users from double-clicking any script files and having them automatically run. To keep your own scripts running, simply associate a new filename extension—such as CORPSCRIPT—with WScript.exe. Name trusted scripts appropriately, and they'll run. It's unlikely a hacker can guess your private filename extension, making this a simple, reasonably effective means of establishing a safer scripting environment.

Script Signing

As I described earlier in this chapter, signing your scripts is a simple and effective way to guarantee their identity. By globally setting the WSH trust policy, you can prevent your computers from running untrusted scripts. There doesn't have to be much expense associated with this technique: You can establish your own Certification Authority (CA) root, use Group Policy to configure all client computers to trust that root, and then use the root to issue yourself a code-signing certificate.

Antivirus Software

Most modern antivirus software watches for script launches and displays some kind of warning message. I don't consider this an effective means of protecting your enterprise from unauthorized scripts; it's difficult to communicate to your users which scripts are "good" and which are "bad," putting them into just as much trouble as before the antivirus solution stepped in to help. However, such software can provide an easy-to-deploy means of protecting against scripts, especially if you aren't planning to use your own scripts on users' machines (as in logon scripts).

Defunct Techniques

Some popular techniques have been used in the past to control scripting that I want to discuss very briefly. I don't consider these methods reliable, secure, or desirable.

- Removing WScript.exe and Cscript.exe. Under Windows 2000 and later, these two files are under Windows File Protection and are not easily removed to begin with. Plus, doing so completely disables scripting, which probably isn't a goal if you're reading this book.
- Disassociating the VBS, WSF, and other filename extensions. Scripts can still be executed by running Wscript.exe scriptname, because that doesn't require a filename extension. In other words, it doesn't require much effort for hackers to e-mail shortcuts that do precisely that, thus defeating this technique as a safety measure.
- Renaming WScript.exe to something else. This is ineffective. Although it prevents the existing file extensions (VBS, etc.) from launching WScript.exe, it doesn't necessarily prevent scripts from running. Additionally, because WScript.exe is under Windows File Protection on Windows 2000 and later, the file may eventually wind up being replaced under your nose.

Review

Scripting can be made safe in almost any environment. The capability of WSH to spot signed scripts and execute them, combined with your ability as an administrator to customize the filename extensions on client and server computers, can provide an effective barrier against unauthorized scripts, still allowing your own scripts to run.

COMING UP

You're all finished! If you've read this book straight through, you've learned how to program in VBScript, use ADSI and WMI, create administrative Web pages, and even secure your environment for safer scripting. In the next part, I'll wrap up with some longer examples of administrative scripts that you can use as references or start running in your environment right away.

Part VI: Ready-to-Run Examples

Chapter 29. Logon and Logoff Scripts

IN THIS CHAPTER

Both NT and Active Directory domains allow you to specify logon scripts; Active Directory also allows you to specify a logoff script for users and computers. I'll provide some sample logon and logoff scripts that you can use as a starting point for building your own.

Perhaps one of the most common uses for scripting is the creation of logon (and, for Active Directory domains, logoff) scripts. A number of scripting languages have been created almost exclusively for use in these scripts, including Microsoft's unsupported Kixtart, the more general-purpose WinBatch, and many others. Dozens of command-line utilities exist that allow batch files to stand in as logon scripts. Although VBScript has a steeper learning curve than these other products, it also offers unmatched power and flexibility. VBScript's capability to use COM objects and directly access many operating system features allows it to immediately take advantage of new technologies and techniques.

Because every environment requires a unique logon script, it's impossible to offer examples that you can truly use without modification. Instead, I've tried to create examples that are modular, allowing you to pick and choose the various tasks you need for your own logon scripts. As a result, some of the tasks my examples perform are slightly less than real world. For example, you'll see examples where I'm using script to execute a relatively useless command-line utility. The point of the example isn't the utility itself, but rather the ability to execute external commands. You should be able to quickly modify the pieces of these examples to assemble your own highly useful scripts.

NOTE

I'm assuming that you know how to designate logon (and logoff) scripts for whatever domain environment you're working in. If you don't, consult the operating system's documentation for more information.

NT and Active Directory Logon Scripts

The first example script works in either an Active Directory (AD) or NT domain environment. It includes a number of common logon script tasks.

NOTE

One thing to be careful of: Windows 9x scripts actually execute before the operating system finishes the user logon process. As a result, the technique I use to retrieve the current user's name won't always work properly. There's no pretty workaround for this; I'll show you one example of how to make your script essentially sit and wait until Windows finishes and the user name becomes available.

➤➤ Logon Script One

Listing 29.1 shows the script code. I've included comments to help identify each task, so that you can easily break this apart and reuse various bits in your own scripts.

Listing 29.1. Logon1.vbs. **This script includes most common logon script tasks.**

```
' sample logon script


' first let's create the objects we'll be using

dim objShell, objNetwork

set objShell = WScript.CreateObject("WScript.Shell")

set objNetwork = WScript.CreateObject("WScript.Network")


' let's display a welcome message

dim strDomain, strUser

strDomain = objNetwork.UserDomain

strUser = objNetwork.UserName

msgbox "Welcome to the " & strDomain & ", " & strUser & "!"




'we'll map the Z: drive to a network location

objNetwork.MapNetworkDrive "Z:", "\\Server\Share"




'let's connect to a network printer and make it

' the default - we'll capture LPT2:

objNetwork.AddPrinterConnection "LPT2", "\\Server\Print1"
```

Active Directory–Specific Logon Scripts

If you're in an AD domain, you can take advantage of AD's newer technologies and built-in scripting interfaces, such as ADSI, to perform more powerful and flexible tricks in your logon scripts.

➤➤ AD Logon Script

Listing 29.3 shows a sample logon script designed to run within an AD environment.

Listing 29.3. ADLogon1.vbs. **This script requires Active Directory to run.**

```
Const G_SALES = "cn=sales"

Const G_MARKETING = "cn=marketing"

Const G_EXECS = "cn=executives"


Set oNetwork = CreateObject("WScript.Network")

oNetwork.MapNetworkDrive "h:", "\\FileServer\Users\" & _

 oNetwork.UserName

Set oADSystemInfo = CreateObject("ADSystemInfo")

Set oUser = GetObject("LDAP://" & oADSystemInfo.UserName)

sGroups = LCase(Join(oUser.MemberOf))


If InStr(sGroups, G_SALES) Then

 oNetwork.MapNetworkDrive "S:", "\\FileServer\SalesDocs\"

 oNetwork.AddWindowsPrinterConnection "\\PrintServer\Quotes"

 oNetwork.SetDefaultPrinter "\\PrintServer\Quotes"

End If


If InStr(sGroups, G_MARKETING) Then

 oNetwork.MapNetworkDrive "M:", "\\FileServer\MarketingDocs\"

 oNetwork.AddWindowsPrinterConnection "\\PrintServer\ColorLaser"

 oNetwork.AddWindowsPrinterConnection "\\PrintServer\BWLaser"

 oNetwork.SetDefaultPrinter "\\PrintServer\BWLaser"

End If


If InStr(sGroups, G_EXECS) Then

 oNetwork.MapNetworkDrive "X:", "\\FileServer\ExecDocs\"
```

Active Directory Logoff Scripts

Keep in mind that AD actually offers four types of automated scripts: logon, startup, logoff, and shutdown. Logon scripts execute when a user logs on, whereas startup scripts execute when a computer starts. Startup scripts are a good place to perform configuration changes, such as changing a computer's IP address. Logon scripts, which are what I've shown you so far in this chapter, make changes to the user's environment.

AD also supports logoff scripts, which execute when a user logs off, and shutdown scripts, which execute when a computer shuts down. It's tougher to find practical applications for these scripts, but there definitely are some. For example, you might copy a custom application's database file to a network server, if the server is available when the user logs off. That would provide a convenient, automated backup for laptop users. If you're mapping drive letters and printers in a logon script, you might unmap those in a logoff script. That way, mobile users won't see those resources if they log on to their machines while they are disconnected from the network.

➤➤ Logoff Script

Listing 29.5 shows a sample logoff script that unmaps a network printer, which was mapped in a logon script. Note that I use On Error Resume Next in this script, so that the script doesn't generate an error if the printer isn't already mapped (which would be the case if the user had manually deleted the mapping already). Note that this is essentially a reverse script of Listing 29.4, and undoes everything that script accomplishes.

Listing 29.5. Logoff.vbs. **This script is designed to run when a user logs off his computer.**

```
Dim oSystemInfo

Dim oShell

Dim sLogonServer, sSiteName


'get logon server

Set oShell = Wscript.CreateObject("Wscript.Shell")

sLogonServer = oShell.ExpandEnvironmentStrings("%LOGONSERVER%")


'get AD site name

Set oSystemInfo = CreateObject("ADSystemInfo")

sSiteName = oSystemInfo.SiteName

'turn off error checking

On Error Resume Next


'unmap printer based on site

Select Case sSiteName

 Case "Boston"

  oNetwork.RemovePrinterConnection "\\BOS01\Laser1"

Case "New York"
```

Review

You've seen several examples of how logon (and logoff) scripts can be used in both AD and NT domains to provide automated client computer configuration. Don't forget, though, that servers are computers, too; using startup and shutdown scripts can be a great way to start third-party utilities on servers, collect software or hardware inventory information, and so forth. In any case, VBScript provides the flexibility and power you need to perform just about any task automatically at startup, logon, logoff, and shutdown.

COMING UP

In the next chapter, I'll provide some example domain administration scripts. You'll see how VBScript can be used to manage users, groups, and domains, and how you can automate processes like adding new users. After that, I'll show you some general network administration samples, and in Chapter 32 show you some all-purpose ADSI and WMI scripts that you can use to start writing your own scripts.

Chapter 30. Windows and Domain Administration Scripts

IN THIS CHAPTER

In this chapter, I provide a few Windows and domain administration scripts, along with the line-by-line explanations I've used throughout this book. These samples are intended to be immediately useful in your own environment, and they provide a great way to see specific scripting features in action so that you can use them more effectively in your own scripts.

A number of different tasks exist within a domain that you may want to automate. Some that pop into mind are automating the process of creating new user accounts, finding users who haven't logged on in a long time and disabling their accounts, and collecting information from the computers in your domain. Whatever your needs, scripting is an excellent solution, and the three sample scripts in this chapter should give you a good idea of what you can accomplish.

Automating User Creation

In this example, I'll show you how to use ActiveX Data Objects (ADO) to query information from an Excel spreadsheet, put that information into script variables, and use those variables to create and configure new domain user objects.

NOTE

I've not covered ADO, and I find it doesn't come up often in many administrative scripts. I don't provide a comprehensive explanation of it here, but this example should give you a starting point if you have a need for a similar script in the future.

To run this script, you're going to need to create an Excel spreadsheet. Leave the first sheet named Sheet1, which is the default, and enter the following column headers on row 1:

- UserID
- FullName
- Description
- HomeDirectory
- Groups
- DialIn

Populate the remaining rows as follows.

- UserID: Enter the unique user ID you want this user to have. Note that the script doesn't do any error checking, and Windows lets you create users with duplicate names in a script. Be careful, though, because user accounts with duplicate names don't behave properly.
- FullName: The full name of the user.
- Description: Optionally, a description of the user.
- HomeDirectory: This needs to be a subfolder under a file server's root folder. You'll see how this gets used later.
- Groups: A comma-delimited list of groups the user should be placed in.
- DialIn: "Yes" or "No" describing whether the user should have dial-in permissions.

Finding Inactive Users

This is a script I like to run from time to time, just to find out how many user accounts haven't logged on for a while. Often, they're accounts of employees who have left, but another administrator (certainly not me!) forgot to remove the accounts. Because the accounts represent a potential security threat, I like to disable them until I can figure out if they're still needed for something.

NOTE

This script works reliably only in Active Directory domains that use Windows Server 2003 domain controllers. Unfortunately, the attribute in Active Directory that stores the last logon date is not reliably updated and replicated in NT or Windows 2000 domains. The only way to use this script in older domains is to run it independently against each domain controller and then compare the results, because each domain controller maintains an independent list of last logon times.

➤➤ Finding Inactive Users

Listing 30.2 demonstrates how to use ADSI to locate users who haven't logged on in a while.

Listing 30.2. FindOldUsers.vbs. **This script checks the LastLogin date to see when users last logged into the domain.**

```
Dim dDate

Dim oUser

Dim oObject

Dim oGroup

Dim iFlags

Dim iDiff

Dim iResult

Const UF_ACCOUNTDISABLE = &H0002


'Set this to TRUE to enable Logging only mode -

'no changes will be made

CONST LogOnly = TRUE


'Point to oObject containing users to check

Set oGroup = _

 GetObject("WinNT://MYDOMAINCONTROLLER/Domain Users")

On error resume next

For each oObject in oGroup.Members
```

Collecting System Information

Software like Microsoft Systems Management Server (SMS) does a great job of collecting information from all of the computers in your environment. However, it's an expensive, complicated product, and sometimes you might just need a quick-and-dirty means of collecting the same information. This script is a great starting point for an inventory collection system that you can make a part of your users' logon scripts.

➤➤ Collecting System Information

Listing 30.3 shows how a WMI script can be used to inventory information from a computer. For example, you could modify this script to run against multiple machines at once, letting you know what servers are running particular types of hardware.

Listing 30.3. CollectSysInfo.vbs**. This script inventories a computer and displays the information.**

```
Set oSystemSet = _

GetObject("winmgmts:").InstancesOf("Win32_ComputerSystem")


For Each oSystem in oSystemSet

 system_name = oSystem.Caption

 system_type = oSystem.SystemType

 system_mftr = oSystem.Manufacturer

 system_model = oSystem.Model

Next

Set oProcSet = _

GetObject("winmgmts:").InstancesOf("Win32_Processor")


For Each oSystem in oProcSet

 proc_desc = oSystem.Caption

 proc_mftr = oSystem.Manufacturer

 proc_mhz = oSystem.CurrentClockSpeed

Next


Set oBiosSet = _

 GetObject("winmgmts:").InstancesOf("Win32_BIOS")


For Each oSystem in oBiosSet

      bios_info = oSystem.Version

Next
```

Review

Managing domains and Windows by using scripts is an effective, efficient use of your VBScript skills. You'll probably find that a good half of the scripts you write, in fact, are designed for Windows or domain management, since those tasks are most often in need of automation. The samples in this chapter provide a great jump-start for improving your environment's security, consistency, and maintainability, all with a few lines of script code!

COMING UP

In Chapter 32, I'll present several network administration scripts. These scripts make heavy use of complicated WMI and ADSI queries, making them great examples for getting started on your own high-end administrative scripts.

Chapter 31. Network Administration Scripts

IN THIS CHAPTER

These are among the most common scripts you'll probably write and run, so I've assembled quite a few ready-to-run examples. You can use these in your environment right now, or you can modify them for your own purposes. I've included line-by-line explanations for each so that you can figure out what they do.

Administrative scripts can be some of the most useful tools in your administrator's toolbox. Perhaps the scripts automate some repetitive task; perhaps they enable you to remotely accomplish tasks that would otherwise require a visit to a user's desktop; or, perhaps they simply allow you to accomplish something that would otherwise be too difficult. In any case, the examples in this chapter cover a wide range of uses, and should give you a better idea of what scripts can help you accomplish.

Shutting Down Remote Computers

This is always a useful trick to have up your sleeve. After you've figured out how to do it, you can perform a number of other useful tricks with remote computers.

➤➤ Shutting Down Remote Computers

Listing 31.1 shows the basic script. You are prompted for a computer name, and then that computer is shut down. This script does use WMI, so both your computer and the one you're shutting down must support WMI, and your user credentials must be accepted on the remote machine as an administrator.

Listing 31.1. Shutdown.vbs. **Shuts down a remote computer by using WMI.**

```
'get machine to shut down

Dim sMachine

sMachine = InputBox("Shut down what computer?")


'create WMI query

Dim sWMI

sWMI = "SELECT * FROM Win32_OperatingSystem WHERE" & _

 "Primary = True"


'Contact specified machine

Dim oOS

Set oOS = GetObject("winmgmts://" & sMachine & _

 "/root/cimv2".ExecQuery(sWMI)


'run through all returned entries

Dim oItem

For Each oItem in oOS

 oItem.Shutdown

Next
```

You don't need to make any changes to this script to get it to run.

➤➤ Shutting Down Remote Computers—Explained

This script is typical of most WMI scripts you've seen, except that it uses a method of the queried WMI instance instead of simply querying information. The script starts by getting the name of the computer you want to work with.

Listing Remote Shares

Ever wonder what shares are available on a remote file server? I've often needed a complete list. Yes, you can use NET VIEW or another command-line utility, but what if you want to list several servers at once, or have the list of shares output to a text file, or used by another script? Having a script capable of generating this list can be a handy tool.

➤➤ Listing Shares

Listing 31.2 shows how to pull a list of shares from any remote computer, using ADSI.

Listing 31.2. Shares.vbs. **Listing remote shares.**

```
sServerName = _

 InputBox("Enter name of server to list shares for.")


Set oFS = GetObject("WinNT://" & sServerName & _

 "/LanmanServer,FileService")

For Each sSh In oFS

      WScript.Echo sSh.name

Next
```

Not very complicated, is it? That's the power of ADSI. You shouldn't have to make any changes to run this script, and it will run on NT, 2000, XP, and 2003 systems.

➤➤ Listing Shares—Explained

This script starts out by simply asking for the name of the server that you want to list shares for.

```
sServerName = InputBox("Enter name of server to list shares for.")
```

Next, it queries ADSI. The ADSI query connects to the specified server's LanManServer, which is a file service. Physically, the Server service is present on all Windows NT-based computers, including NT, 2000, XP, and 2003.

```
Set oFS = GetObject("WinNT://" & sServerName & _

 "/LanmanServer,FileService")
```

The file service has a collection of shares, and this next loop simply iterates each of them and displays each in a message box (or outputs to the command line, if you're running through Cscript.exe).

```
For Each sSh In oFS

      WScript.Echo sSh.name
```

Finding Out Who Has a File Open

➤➤ Who Has a File

Listing 31.4 shows the script.

Listing 31.4. WhoHas.vbs. **Shows who has a particular file open.**

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

' get the local path of the file to check
varFile= _
 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 objFS = GetObject("WinNT://" & varServer & _
 "/lanmanserver,fileservice")

' scan through the open resources until we
' locate the file we want
varFoundNone = True

' use a FOR...EACH loop to walk through the
' open resources
For Each objRes in objFS.Resources

      ' does this resource match the one we're looking for?
      If objRes.Path = varFile Then

            ' we found the file - show who's got it
            varFoundNone = False
            WScript.Echo objRes.Path & " is opened by " & _
             objRes.User
```

Uninstall Remote MSI Packages

Using WMI to interact with MSI packages seems tricky, but it's not too complicated. Wouldn't it be nice to have a script that shows you which MSI packages are installed on a remote computer, and lets you selectively uninstall one? You could remotely weed out unapproved applications on users' machines, maintain servers, and a host of other useful tasks.

➤➤ Remote MSI Uninstall

Listing 31.5 shows the script. It prompts you for a machine name, and then shows you which packages are installed. Note that the one thing this script doesn't do is work against the machine it's running on; that's because WMI doesn't allow you to specify alternate user credentials when accessing the local machine. If you want to uninstall something locally, use the Control Panel!

NOTE

This script runs on Windows NT 4.0, Windows XP, and Windows 2000. However, Windows Server 2003 requires the optional Windows Installer provider, which is included on the Windows Server 2003 CD-ROM.

Listing 31.5. Uninstall.vbs. **Uninstalls a remote MSI package.**

```
'get remote computer name

Dim sMachine

sMachine = InputBox("Computer name?")


'get admin credentials

Dim sAdminUser, sPassword

sAdminUser = InputBox("Enter the admin user name.")

sPassword = InputBox("Enter the users password. ")


'get a WMI Locator

Dim oLocator

Set oLocator = CreateObject("WbemScripting.SWbemLocator")


'connect to remote machine

Dim oService

Set oService = oLocator.ConnectServer(sMachine, "root\cimv2", _

    sAdminUser, sPassword)


'get a list of installed products

Dim sMsg, sName
```

Adding Users from Excel

Adding users to your domain in bulk can be a great timesaver, especially in rapidly growing organizations. This example lists users in an Excel spreadsheet, and actually uses ActiveX Data Objects (ADO) to read information from the spreadsheet.

You need to do a bit of preparation to use this script. First, create a new Excel spreadsheet. The spreadsheet should look something like Table 31.1.

Notice that the Groups column is a comma-delimited list of group names, with no spaces after the commas.

Table 31.1. Sample Excel Spreadsheet for Adding Users

UserID	FullName	Description	Home Directory	Groups	DialIn
DonJ	Don Jones	Administrator	Donj	Domain Admins	Y
GregM	Greg Marino	Sales	Gregm	Sales,Execs	N

You also need to create an ODBC DSN (Data Source Name) that points to the spreadsheet. On Windows XP, open the Administrative Tools program group, and select the Data Sources item. Create a new System DSN that uses the Microsoft Excel driver. Name the DSN "Excel." When you make the Excel spreadsheet, be sure to create your columns and rows on Sheet1, and don't change the sheet name.

➤➤ Adding Users

Listing 31.6 shows the complete script, which uses the Excel sheet and DSN you set up earlier.

Listing 31.6. AddUsers.vbs. **Adds users in bulk from an Excel spreadsheet.**

```
' 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://NT4PDC")
```

Listing Hot Fixes and Software

Wouldn't it be nice to have a script that you could run on each computer in your enterprise to get an inventory of hot fixes and software applications? It's not hard! Rather than showing you a single sample script, though, I want to walk through this example a bit more modularly. The first thing I need is a routine that determines the local computer's name, and then opens an output text file on a file server somewhere.

```
Dim oNetwork

Set oNetwork = CreateObject("WScript.Network")


Dim sLocal

sLocal = oNetwork.ComputerName


Dim oFSO, oTS

Set oFSO = CreateObject("Scripting.FileSystemObject")

Set oTS = oFSO.CreateTextFile("\\server\" & _

 sLocal & ".txt")
```

This results in an object oTS, which is a TextStream object representing an output text file. The file is named after the computer on which it runs, and you can modify the location to be a file server in your environment.

I just need to find a list of hot fixes and applications, and I don't need to turn any further than the Scriptomatic tool, or the WMI Query Wizard in PrimalScript. Hot fixes are formally known as QFEs, or Quick Fix Engineering patches, and there's a WMI class just for them. The following wizard-generated code queries it for me.

```
On Error Resume Next

Dim strComputer

Dim objWMIService

Dim colItems


strComputer = "."

Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\root\cimv2")

Set colItems = objWMIService.ExecQuery("Select * from Win32_QuickFixEngineering",,48)

For Each objItem in colItems

 WScript.Echo "Caption: " & objItem.Caption

 WScript.Echo "CSName: " & objItem.CSName

 WScript.Echo "Description: " & objItem.Description

 WScript.Echo "FixComments: " & objItem.FixComments
```

Review

Administration can be faster and easier with scripting in your bag of tricks. In this chapter, you've seen how to combine basic VBScript, the FileSystemObject, WMI, ADSI, and other components to create effective, automated administration tools. This is truly the essence of administrative scripting: gluing together these various components with the help of VBScript to create tools you'll use over and over again.

COMING UP

In the last chapter of this book, I'll present more examples of scripts that utilize WMI and ADSI. Although you've already seen several examples that use WMI and ADSI, they can be difficult-to-approach technologies, so I figure a few more examples can't hurt.

Chapter 32. WMI and ADSI Scripts

IN THIS CHAPTER

Although I've shown a dozen example scripts for WMI and ADSI, you may still feel like both technologies are a bit complex—they certainly look complex. My goal here is to provide some examples that will help take the mystery out of them.

WMI and ADSI both come across as incredibly complex. It's not surprising; the official Microsoft documentation doesn't help make them any more approachable. However, I've found that a few generic WMI and ADSI scripts can show you how to do just about anything with either of the two technologies. That's what this chapter is all about: providing you with some templates that you can use to write your own WMI and ADSI scripts to do anything you need.

The All-Purpose WMI Query Script

About 75% of my time with WMI is spent querying information. Fortunately, there's a very simple template you can use. In fact, this is identical to the code produced by PrimalScript's WMI Query Wizard and by the Microsoft Scriptomatic tool. The fact that those tools exist goes to show how generic and all-purpose WMI scripting really can be.

Start by finding the WMI class that you need to query. This isn't hard, but it can be time-consuming because there are so many classes to choose from. I usually use PrimalScript or the Scriptomatic to browse through the classes until I see one I want.

Next, define the three variables you need to query WMI.

```
Dim strComputer

Dim objWMIService

Dim colItems
```

Then, write the actual query.

```
strComputer = "."

Set objWMIService = GetObject("winmgmts:\\" & _

 strComputer & "\root\cimv2")

Set colItems = objWMIService.ExecQuery( _

 "Select * from class_name_here",,48)
```

Notice that you can change the value assigned to strComputer to query a remote machine. Insert the appropriate class name. Next, write a For Each…Next loop that iterates through the classes that you queried.

```
For Each objItem in colItems

Next
```

Finally, insert the appropriate lines within the construct to work with the class' properties.

```
WScript.Echo objItem.property_name_here

WScript.Echo objItem.property_name_here
```

If you don't want to display the information, write it to a text file, assign it to a variable, or do whatever you like. For example, let's say you want to limit the number of class instances returned by your query. When querying Win32_QuickFixEngineering, you receive all installed hot fixes. What if you just want a particular one? No problem. Modify your query appropriately.

The All-Purpose WMI Update Script

Querying is easy, but what about updating information in WMI? Just as easy. Start with the basic query template. Instead of echoing property information, however, simply use one of the class' methods, which are documented in the MSDN Library. Here's how to find the Library.

1. Go to http://msdn.microsoft.com/library.
2. In the left-hand menu tree, expand Setup and System Administration.
3. Expand Windows Management Instrumentation.
4. Expand SDK Documentation.
5. Expand WMI Reference.
6. Expand WMI Classes.
7. Expand Win32 Classes.
8. Expand the appropriate category, such as Computer System Hardware Classes, for the class you're interested in.
9. Expand the class itself for a listing of methods, or click on the class for a list of properties.

Take the Win32_NetworkAdapterConfiguration class as an example. This class has a property named DHCPEnabled, which reads True or False. It seems that setting this to True would enable DHCP, but reading the documentation indicates that this particular property is read-only. However, expanding the class definition shows a method named EnableDHCP that looks like just the thing. The following script would enable DHCP on the network adapters in a computer.

```
Dim strComputer

Dim objWMIService

Dim colItems



strComputer = "."

Set objWMIService = GetObject( _

 "winmgmts:\\" & strComputer & "\root\cimv2")

Set colItems = objWMIService.ExecQuery( _

 "Select * from Win32_NetworkAdapterConfiguration",,48)
```

The All-Purpose ADSI Object Creation Script

Creating objects in ADSI doesn't require a lot of code, either. Here's a generic, all-purpose script for creating a user.

```
strContainer = ""
strName = "UserName"

'generic part
Set objRootDSE = GetObject("LDAP://rootDSE")
If strContainer = "" Then
  Set objContainer = GetObject("LDAP://" & _
    objRootDSE.Get("defaultNamingContext"))
Else
  Set objContainer = GetObject("LDAP://" & _
    strContainer & "," & _
    objRootDSE.Get("defaultNamingContext"))
End If

'create the object
Set objUser = objContainer.Create("user", "cn=" & strName)
objUser.Put "sAMAccountName", strName
objUser.SetInfo
```

This script is pretty much exactly what the Microsoft EZADScriptomatic tool generates for you. Just fill in the container name with an organizational unit (OU) name or some other container name; otherwise, the user is created in the default location for your domain. Do you need to create a computer instead? Just change the last three lines of code to read like this.

```
Set objComputer = objContainer.Create("computer", "cn=" & strName)
objComputer.SetInfo
```

Do you want a new contact object? Here are the last couple of lines of code.

```
Set objContact = objContainer.Create("contact", "cn=" & strName)
objContact.SetInfo
```

The All-Purpose ADSI Object Query Script

Querying information is just as easy. Your script starts out with some generic object-retrieval code.

```
strContainer = ""

strName = "UserName"


'generic part

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
```

Then, you add code depending on what type of object you're querying. For example, to get a user's name, just access objItem.Get("Name"). For the user's display name, it's just objItem.Get("displayName"). For an exhaustive list of properties of users, contacts, groups, computers, and organizational units, see the ADSI documentation, or just use the EZADScriptomatic, which you can download from www.microsoft.com/scripting.

The All-Purpose ADSI Object Deletion Script

This template script looks a lot like the object creation script to start with.

```
strContainer = ""

strName = "UserName"


'generic part

Set objRootDSE = GetObject("LDAP://rootDSE")

If strContainer = "" Then

  Set objContainer = GetObject("LDAP://" & _

    objRootDSE.Get("defaultNamingContext"))

Else

  Set objContainer = GetObject("LDAP://" & _

    strContainer & "," & _

    objRootDSE.Get("defaultNamingContext"))

End If
```

This code simply connects to a container, such as an OU. After you're connected, you can delete the object, such as a user.

```
objContainer.Delete "user", "cn=" & strName
```

You can reuse this line of code to delete a contact, computer, or group just by replacing "user" with "contact", "computer", or "group" as appropriate. Nothing could be simpler.

Mass Password Changes with ADSI

One cool use for ADSI that folks don't often think of is using it to manage the local Security Accounts Manager (SAM) of member computers. I have a useful script I run every 30 days or so to change the local Administrator passwords on all my machines; I use the same script to also change some other special user accounts I've created.

➤➤ Mass Password Changes

Listing 32.1 shows the script. Note that it reads the computer names from a text file, which lists one computer name per line. This way, I just have to maintain the text file list. You could also write the script to first query all of the computer names in the domain if you want a higher level of automation with less maintenance.

Listing 32.1. MassPass.vbs. **Changes local Administrator passwords.**

```
Dim oFSO, oTSIn

Dim sComputer, sUser, oUser, sDSPath

Dim sNewPass


sUser = "Administrator"

sNewPass = "pA55w0Rd!"


Set oFSO = CreateObject("Scripting.FileSystemObject")

Set oTSIn = oFSO.OpenTextFile("c:\machines.txt")


Do Until oTSIn.AtEndOfStream

 sComputer = oTSIn.ReadLine

 sDSPath = "WinNT://" & sComputer & "/" & sUser & ",user"


 Set oUser = GetObject(sDSPath)

  If Err Then

   WScript.Echo sComputer & " could not be contacted"

   Err.Clear

  Else

   oUser.SetPassword newPassword

  End If

Loop

MsgBox "Complete"

oTSIn.Close
```

Review

Hopefully, the examples in this chapter—particularly the generic WMI and ADSI scripts—will help you conquer any lingering fears or misgivings about becoming a power scripter. Both WMI and ADSI are incredibly useful technologies, and with the examples I've provided in this chapter and elsewhere in this book, you should be able to do just about anything you want with them.

ALL FINISHED!

Welcome to the end! I've provided you with an appendix that you can use to learn more about scripting technologies, and to help you as a quick reference for future scripts. However, you should be ready to start scripting on your own. Have fun, experiment a lot, and don't be afraid to steal ideas or code from the examples in this book! Good luck!

Part VII: Appendix

Appendix Administrator's Quick Script Reference

One of the toughest parts about scripting, at least for administrators, can be figuring out which VBScript command or object to use for a particular purpose. I've created this appendix to help with that problem. If you need to do something in particular with VBScript, but don't know exactly how, this appendix is where you need to be. Look up the task you're trying to perform, and I'll provide you with the specific VBScript statement, function, object, or other element that you need to use. I'll also provide you with a cross-reference to the chapter in this book that covers the particular element. Don't forget that you can also use Google and other Web-based resources to figure out how to perform specific tasks.

I cover some tips for using Google as a scripting resource in Chapter 4. Here's a sample entry:

Files, working with: FileSystemObject (12)

This entry indicates that you can use the FileSystemObject to work with files, and that you can find more information about the FileSystemObject in Chapter 12. I don't provide a chapter for every feature, generally because I don't cover them all in detail. When there's no chapter reference, simply refer to the feature's documentation in the main VBScript documentation. Sometimes, I provide a chapter reference when I've discussed similar features. For example, if you look up "Rounding" in this index you'll find a reference to Chapter 7. I don't specifically discuss rounding in Chapter 7, but I do discuss similar functions. In this index, I provide the name of appropriate VBScript functions—such as Round()—and you can look them up in Microsoft's documentation. The idea is just to give you a pointer to the functions and objects you'll need, so that you don't have to wander aimlessly through the documentation.

I've tried to list each feature with as many descriptions as I could think of. For example, if you're trying to figure out a way to trap or handle errors, you could look for "Trapping," "Handling," or just "Errors" and find what you're looking for.

NOTE

For Windows Management Instrumentation (WMI), I've also provided the name of appropriate WMI classes in italics. That will direct you to the appropriate class reference in the WMI documentation more quickly.

1394: WMI (17-19) Win32_1394Controller, Win32_1394ControllerDevice

A

B

C

D

E

F

G

H

I

Input devices: WMI (17–19) Win32_Keyboard, Win32_PointingDevice

Input, user: InputBox (6)

J

K

L

M

N

O

P

Q

R

S

T

Telephony: WMI (17–19) Win32_POTSMOdem, Win32_POTSModemToSerialPort

Text files: FileSystemObject (12)

Time and Date functions: DateAdd, DateDiff, DatePart, DateValue, Day, Month, MonthName, Weekday, WeekdayName, Year, Hour, Minute, Second, Now (9)

Time calculations: DateAdd, DateDiff (9)

Time, retrieving: Time (9)

Trapping errors: On Error

Trimming strings: Trim, LTrim, RTrim (8)

Type of variables: IsArray, IsDate, IsEmpty, IsNull, IsNumeric, IsObject

U

URL shortcuts: WSH UrlShortcut object (11)

USB: WMI (17–19) Win32_USBController, Win32_USBControllerDevice, Win32USBHub

User input: InputBox (6)

User name: WSH Network object (11)

Users and groups: WMI (17–19) Win32_Account, Win32_Group, Win32_GroupInDomain, Win32_GroupUser, Win32_LogonSession, Win32_LogonSessionMappedDisk, Win32_NetworkLoginProfile, Win32_SystemAccount, Win32_UserAccount, Win32_UserInDomain

Users, local: WMI (17–19) Win32_SystemUsers

Users, working with: ADSI (14–16)

V

W

CD-ROM Warranty

Addison-Wesley warrants the enclosed CD-ROM to be free of defects in materials and faulty workmanship under normal use for a period of ninety days after purchase (when purchased new). If a defect is discovered in the CD-ROM during this warranty period, a replacement CD-ROM can be obtained at no charge by sending the defective CD-ROM, postage prepaid, with proof of purchase to:

Disc Exchange

Addison-Wesley Professional

Pearson Technology Group

75 Arlington Street, Suite 300

Boston, MA 02116

Email: AWPro@aw.com

Addison-Wesley makes no warranty or representation, either expressed or implied, with respect to this software, its quality, performance, merchantability, or fitness for a particular purpose. In no event will Addison-Wesley, its distributors, or dealers be liable for direct, indirect, special, incidental, or consequential damages arising out of the use or inability to use the software. The exclusion of implied warranties is not permitted in some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights. There may be other rights that you may have that vary from state to state. The contents of this CD-ROM are intended for personal use only.

More information and updates are available at:

http://www.awprofessional.com/