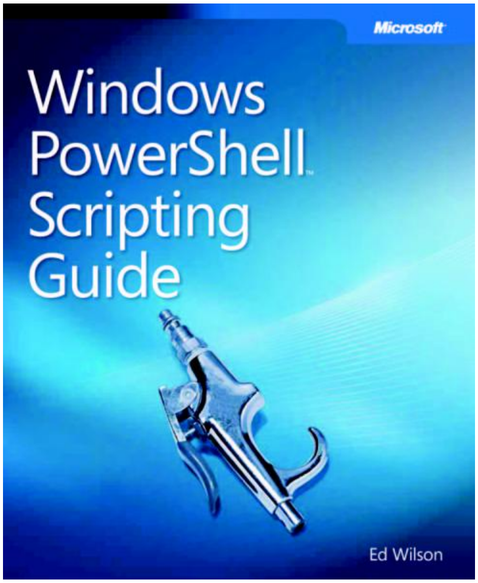

Windows PowerShell™ Scripting Guide

Ed Wilson

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/9541.aspx

9780735622791

Table of Contents

Acknowledgments . . . . . *xvii*
Introduction . . . . . *xix*
Is This Book for Me? . . . . . *xix*
About the Companion CD . . . . . *xx*
System Requirements . . . . . *xxi*
Technical Support . . . . . *xxi*

1 The Shell in Windows PowerShell **1**
Installing Windows PowerShell 1
Verifying Installation with VBScript 1
Deploying Windows PowerShell 2
Interacting with the Shell 3
Introducing Cmdlets 5
Configuring Windows PowerShell 6
Creating a Windows PowerShell Profile 6
Configuring Windows PowerShell Startup Options 6
Security Issues with Windows PowerShell 7
Controlling the Execution of Cmdlets 7
Confirming Commands 9
Suspending Confirmation of Cmdlets 10
Supplying Options for Cmdlets 11
Working with Get-Help 12
Working with Aliases to Assign Shortcut Names to Cmdlets 15
Additional Uses of Cmdlets 16
Using the Get-ChildItem Cmdlet 17
Formatting Output 17
Using the Get-Command Cmdlet 24

Exploring with the Get-Member Cmdlet 27
Summary 31

2 Scripting Windows PowerShell 33
Why Use Scripting? 33
Configuring the Scripting Policy 36
Running Windows PowerShell Scripts 39
Use of Variables 39
Use of Constants 40
Using Flow Control Statements 41
Adding Parameters to ForEach-Object 42
Using the *Begin* Parameter 42
Using the *Process* Parameter 43
Using the *End* Parameter 43
Using the *For* Statement 43
Using Decision-Making Statements 44
Using *If ... Elseif ... Else* 45
Using *Switch* 46
Working with Data Types 49
Unleashing the Power of Regular Expressions 53
Using Command-Line Arguments 56
Summary 58

3 Managing Logs 59
Identifying the Event Logs 59
Reading the Event Logs 60
Exporting to Text 61
Export to XML 62
Perusing General Log Files 64
Examining Multiple Logs 65
Retrieving a Single Event Log Entry 66
Searching the Event Log 68
Filtering on Properties 69
Selecting the Source 69
Selecting the Severity 70
Selecting the Message 70

Managing the Event Log 71
Identifying the Sources. 71
Modifying the Event Log Settings. 71
Examining WMI Event Logs 75
Making Changes to the WMI Logging Level. 76
Using the Windows Event Command-Line Utility 76
Writing to Event Logs 77
Creating a Source 77
Putting Cmdlet Output into the Log 78
Creating Your Own Event Logs 79
Summary. 80

4 Managing Services 81

Documenting the Existing Services 81
Working with Running Services 82
Writing to a Text File 83
Writing to a Database 85
Setting the Service Configuration. 94
Accepting Command-Line Arguments 97
Stopping Services 97
Performing a Graceful Stop 99
Starting Services 101
Performing a Graceful Start 102
Desired Configuration Maintenance 107
Verifying Desired Services Are Stopped 108
Reading a File to Check Service Status. 109
Verifying Desired Services Are Running 110
Confirming the Configuration. 110
Producing an Exception Report 111
Summary. 113

5 Managing Shares. 115

Documenting Shares. 115
Documenting User Shares 122
Writing Shares to Text. 125
Documenting Administrative Shares 126
Writing Share Information to a Microsoft Access Database 126

Auditing Shares 130
Modifying Shares 133
Using Parameters with the Script 134
Translating the Return Code 135
Creating New Shares 137
Creating Multiple Shares 141
Deleting Shares 143
Deleting Only Unauthorized Shares 145
Summary 146

6 Managing Printing 147

Inventorying Printers 147
Querying Multiple Computers 148
Logging to a File 150
Writing to a Microsoft Access Database 152
Reporting on Printer Ports 157
Identifying Printer Drivers 163
Installing Printer Drivers 165
Installing Printer Drivers Found on Your Computer 165
Installing Printer Drivers Not Found on Your Computer 167
Summary 169

7 Desktop Maintenance 171

Maintaining Desktop Health 171
Inventorying Drives 171
Writing Disk Drive Information to Microsoft Access 175
Working with Partitions 179
Matching Disks and Partitions 181
Working with Logical Disks 184
Monitoring Disk Space Utilization 188
Logging Disk Space to a Database 192
Monitoring File Longevity 196
Monitoring Performance 199
Using Performance Counter Classes 200
Identifying Sources of Page Faults 204
Summary 204

8 Networking 207

Working with Network Settings 207

Reporting Networking Settings. 207

Working with Adapter Configuration. 212

Filtering Only Properties that Have a Value 218

Configuring Network Adapter Settings 223

Detecting Multiple Network Adapters 223

Writing Network Adapter Information to a Microsoft Excel Spreadsheet . . . 224

Identifying Connected Network Adapters. 228

Setting Static IP Address. 230

Enabling DHCP. 235

Configuring the Windows Firewall 239

Reporting Firewall Settings 240

Configuring Firewall Settings. 241

Summary. 243

9 Configuring Desktop Settings. 245

Working with Desktop Configuration Issues. 245

Setting Screen Savers 245

Auditing Screen Savers 246

Listing Only Properties with Values 252

Reporting Secure Screen Savers 256

Managing Desktop Power Settings 263

Changing the Power Scheme 269

Summary. 275

10 Managing Post-Deployment Issues 277

Setting the Time 277

Setting the Time Remotely. 278

Logging Results to the Event Log 283

Configuring the Time Source 289

Using the Net Time Command 290

Querying the Registry for the Time Source. 292

Enabling User Accounts 297

Creating a Local User Account 303

Creating a Local User 303

Creating a Local User Group 306

Configuring the Screen Saver 309
Renaming the Computer 316
Shutting Down or Rebooting a Remote Computer 319
Summary 323

11 Managing User Data 325

Working with Backups 325
Configuring Offline Files 328
Enabling the Use of Offline Files 331
Working with System Restore 340
Retrieving System Restore Settings 340
Listing Available System Restore Points 344
Summary 347

12 Troubleshooting Windows 349

Troubleshooting Startup Issues 349
Examining the Boot Configuration 349
Examining Startup Services 352
Displaying Service Dependencies 355
Examining Startup Device Drivers 360
Investigating Startup Processes 365
Investigating Hardware Issues 368
Troubleshooting Network Issues 373
Summary 377

13 Managing Domain Users 379

Creating Organizational Units 379
Creating Domain Users 382
Modifying User Attributes 385
Modifying General User Information 386
Modifying the Address Tab 387
Modifying the Profile Tab 388
Modifying the Telephone Tab 389
Modifying the Organization Tab 389
Modifying a Single User Attribute 390
Creating Users from a .csv File 393
Setting the Password 394
Enabling the User Account 394

Creating Domain Groups . . . 395
Adding a User to a Domain Group . . . 398
Adding Multiple Users with Multiple Attributes . . . 400
Summary . . . 404

14 Configuring the Cluster Service . . . 405

Examining the Clustered Server . . . 405
Reporting Cluster Configuration . . . 411
Reporting Node Configuration . . . 416
Querying Multiple Cluster Classes . . . 420
Managing Nodes . . . 431
Adding and Evicting Nodes . . . 431
Removing the Cluster . . . 437
Summary . . . 442

15 Managing Internet Information Services . . . 443

Enabling Internet Information Services Management . . . 443
Reporting IIS Configuration . . . 445
Reporting Site Information . . . 445
Reporting on Application Pools . . . 447
Reporting on Application Pool Default Values . . . 451
Reporting Site Limits . . . 454
Listing Virtual Directories . . . 457
Creating a New Web Site . . . 459
Creating a New Application Pool . . . 464
Starting and Stopping Web Sites . . . 467
Summary . . . 471

16 Working with the Certificate Store . . . 473

Locating Certificates in the Certificate Store . . . 473
Listing Certificates . . . 479
Locating Expired Certificates . . . 483
Identifying Certificates about to Expire . . . 488
Managing Certificates . . . 492
Inspecting a Certificate . . . 492
Importing a Certificate . . . 497
Deleting a Certificate . . . 501
Summary . . . 507

17 Managing the Terminal Services Service 509
Configuring the Terminal Service Installation 509
Documenting Terminal Service Configuration 509
Disabling Logons 513
Modifying Client Properties 517
Managing Users 521
Enabling Users to Access the Server 524
Configuring Client Settings 527
Summary 539

18 Configuring Network Services 541
Reporting DNS Settings 541
Configuring DNS Logging Settings 548
Reporting Root Hints 556
Querying "A" Records 557
Configuring DNS Server Settings 562
Reporting DNS Zones 568
Creating DNS Zones 571
Managing WINS and DHCP 576
Summary 581

19 Working with Windows Server 2008 Server Core 583
Initial Configuration 583
Joining the Domain 584
Setting the IP Address 592
Configuring the DNS Settings 597
Renaming the Server 605
Managing Windows Server 2008 Server Core 611
Monitoring the Server 611
Querying Event Logs 614
Summary 617

A Cmdlet Naming Conventions 619

B ActiveX Data Object Provider Names 621

C Frequently Asked Questions 623

D Scripting Guidelines 631

General Script Construction. 631

Include Functions in the Script that Calls the Function 631

Use Full Cmdlet Names and Full Parameter Names. 632

Use Get-Item to Convert Path Strings to Rich Types 633

General Script Readability 633

Formatting Your Code 634

Working with Functions 635

Creating Template Files. 637

Writing Functions 637

Creating and Naming Variables and Constants 638

E General Troubleshooting Tips 639

Index 643

Chapter 1

The Shell in Windows PowerShell

After completing this chapter, you will be able to:

- Install and configure Windows PowerShell.
- Tackle security issues with Windows PowerShell.
- Understand the basics of cmdlets.
- Work with aliases to assign shortcut names to cmdlets.
- Get help using Windows PowerShell.

On the Companion Disc All the scripts used in this chapter are located on the CD-ROM that accompanies this book in the \scripts\chapter01 folder.

Installing Windows PowerShell

Because Windows PowerShell is not installed by default on any operating system released by Microsoft, it is important to verify the existence of Windows PowerShell on the platform before the actual deployment of either scripts or commands. This can be as simple as trying to execute a Windows PowerShell command and looking for errors. You can easily accomplish this from inside a batch file by querying the value %errorlevel%.

Verifying Installation with VBScript

A more sophisticated approach to the task of verifying the existence of Windows PowerShell on the operating system is to use a script that queries the *Win32_QuickFixEngineering* Windows Management Instrumentation (WMI) class. FindPowerShell.vbs is an example of using *Win32_QuickFixEngineering* in Microsoft Visual Basic Scripting Edition (VBScript) to find an installation of Windows PowerShell.

The FindPowerShell.vbs script uses the WMI moniker to create an instance of the *SwbemServices* object and then uses the *execquery* method to issue the query. The WMI Query Language (WQL) query uses the *like* operator to retrieve hotfixes with a hotfix ID such as 928439, which is the hotfix ID for Windows PowerShell on Windows XP, Windows Vista, Windows Server 2003, and Windows Server 2008. Once the hotfix is identified, the script simply prints out the name of the computer stating that Windows PowerShell is installed. This is shown in Figure 1-1.

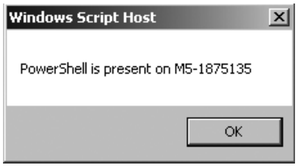

Figure 1-1 The FindPowerShell.vbs script displays a pop-up box indicating that Windows PowerShell has been found.

If the hotfix is not found, the script indicates that Windows PowerShell is not installed. The FindPowerShell.vbs script can easily be modified to include additional functionality you may require on your specific network. For example, you may want to run the script against multiple computers. To do this, you can turn *strComputer* into an array and type in multiple computer names. Or, you can read a text file or perform an Active Directory directory service query to retrieve computer names. You could also log the output from the script rather than create a pop-up box.

FindPowerShell.vbs

```
Const RtnImmedFwdOnly = &h30
strComputer = "."
wmiNS = "\root\cimv2"
wmiQuery = "Select * from win32_QuickFixEngineering where hotfixid like '928439'"

Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.ExecQuery(wmiQuery,,RtnImmedFwdOnly)

For Each objItem in colItems
    Wscript.Echo "PowerShell is present on " & objItem.CSName
Wscript.quit
Next
Wscript.Echo “PowerShell is not installed”
```

Deploying Windows PowerShell

Once Windows PowerShell is downloaded from *http://www.microsoft.com/downloads,* you can deploy Windows PowerShell in your environment by using any of the standard methods you currently use. A few of the methods customers use to deploy Windows PowerShell follow:

- Create a Microsoft Systems Management Server (SMS) package and advertise it to the appropriate organizational unit (OU) or collection.
- Create a Group Policy Object (GPO) in Active Directory and link it to the appropriate OU.
- Call the executable by using a logon script.

If you are not deploying to an entire enterprise, perhaps the easiest way to install Windows PowerShell is to simply double-click the executable and step through the wizard.

Keep in mind that Windows PowerShell is installed by using hotfix technology. This means it is an update to the operating system, and not an add-on program. This has certain advantages, including the ability to provide updates and fixes to Windows PowerShell through operating system service packs and through Windows Update. But there are also some drawbacks, in that hotfixes need to be uninstalled in the same order that they were installed. For example, if you install Windows PowerShell on Windows Vista and later install a series of updates, then install Service Pack 1, and suddenly decide to uninstall Windows PowerShell, you will need to back out Service Pack 1 and each hotfix in the appropriate order. (Personally, at that point I think I would just back up my data, format the disks, and reinstall Windows Vista. I think it would be faster. But all this is a moot point anyway, as there is little reason to uninstall Windows PowerShell.)

Understanding Windows PowerShell

One issue with Windows PowerShell is grasping what it is. In fact, the first time I met Jeffrey Snover, the chief architect for Windows PowerShell, one of the first things he said was, “How do you describe Windows PowerShell to customers?”

So what *is* Windows PowerShell? Simply stated, Windows PowerShell is the next generation command shell and scripting language from Microsoft that can be used to replace both the venerable Cmd.exe command interpreter and the VBScript scripting language.

This dualistic behavior causes problems for many network administrators who are used to the Cmd.exe command interpreter with its weak batch language and the powerful (but confusing) VBScript language for automating administrative tasks. These are not bad tools, but they are currently used in ways that were not intended when they were created more than a decade ago. The Cmd.exe command interpreter was essentially the successor to the DOS prompt, and VBScript was more or less designed with Web pages in mind. Neither was designed from the ground up for network administrators.

Interacting with the Shell

Once Windows PowerShell is launched, you can use it in the same manner as the Cmd.exe command interpreter. For example, you can use *dir* to retrieve a directory listing. You can also use *cd* to change the working directory and then use *dir* to produce a directory listing just as you would perform these tasks from the CMD shell. This is illustrated in the UsingPowerShell.txt example that follows, which shows the results of using these commands.

UsingPowerShell.txt

```
PS C:\Users\edwils> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\edwils
```

```
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d-r--        11/29/2006   1:32 PM            Contacts
d-r--          4/2/2007  12:51 AM            Desktop
d-r--          4/1/2007   6:53 PM            Documents
d-r--        11/29/2006   1:32 PM            Downloads
d-r--          4/2/2007   1:10 AM            Favorites
d-r--          4/1/2007   6:53 PM            Links
d-r--        11/29/2006   1:32 PM            Music
d-r--        11/29/2006   1:32 PM            Pictures
d-r--        11/29/2006   1:32 PM            Saved Games
d-r--          4/1/2007   6:53 PM            Searches
d-r--          4/2/2007   5:53 PM            Videos


PS C:\Users\edwils> cd music
PS C:\Users\edwils\Music> dir
```

In addition to using traditional command interpreter commands, you can also use some of the newer command-line utilities such as Fsutil.exe, as shown here. Keep in mind that access to Fsutil.exe requires administrative rights. If you launch the standard Windows PowerShell prompt from the Windows PowerShell program group, you will not have administrative rights, and the error shown in Figure 1-2 will appear.

Figure 1-2 Windows PowerShell respects user account control and by default will launch with normal user privileges. This can generate errors when trying to execute privileged commands.

Fsutil.txt

```
PS C:\Users\edwils> sl c:\mytest
PS C:\mytest> fsutil file createNew c:\mytest\myNewFile.txt 1000
File c:\mytest\myNewFile.txt is created
PS C:\mytest> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\mytest


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          5/8/2007   7:30 PM       1000 myNewFile.txt


PS C:\mytest>
```

> **Tip** I recommend creating two Windows PowerShell shortcuts and saving them to the Quick Launch bar. One shortcut launches with normal user permissions and the other launches with administrative rights. By default you should use the normal user shortcut and document those occasions that require administrative rights.

When you are finished working with the files and the folder, you can delete the file very easily by using the *del* command. To keep from typing the entire file name, you can use wildcards such as *.txt. This is safe enough, since you have first used the *dir* command to ensure there is only one text file in the folder. Once the file is removed, you can use *rd* to remove the directory. As shown in DeleteFileAndFolder.txt example that follows, these commands work exactly the same as you would expect when working with the command prompt.

DeleteFileAndFolder.txt

```
PS C:\> sl c:\mytest
PS C:\mytest> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\mytest


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          5/8/2007   7:30 PM       1000 myNewFile.txt


PS C:\mytest> del *.txt
PS C:\mytest> cd c:\
PS C:\> rd c:\mytest
PS C:\> dir c:\mytest
Get-ChildItem : Cannot find path 'C:\mytest' because it does not exist.
At line:1 char:4
+ dir  <<<< c:\mytest
PS C:\>
```

With these examples, you have been using Windows PowerShell in an interactive manner. This is one of the primary uses of Windows PowerShell. In fact, the Windows PowerShell team expects that 80 percent of users will work with Windows PowerShell interactively—simply as a better command prompt. You open up a Windows PowerShell prompt and type in commands. The commands can be typed one at a time or they can be grouped together like a batch file. This will be discussed later, as the process doesn't work by default.

Introducing Cmdlets

In addition to using traditional programs and commands from the Cmd.exe command interpreter, you can also use the cmdlets that are built into Windows PowerShell. *Cmdlet* is a name created by the Windows PowerShell team to describe these native commands. They are like executable programs but because they take advantage of the facilities built into Windows

PowerShell, they are easy to write. They are not scripts, which are uncompiled code, because they are built using the services of a special Microsoft .NET Framework namespace. Because of their different nature, the Windows PowerShell team came up with the new term *cmdlet*. Windows PowerShell comes with more than 120 cmdlets designed to assist network administrators and consultants to easily take advantage of Windows PowerShell without having to learn the Windows PowerShell scripting language. These cmdlets are documented in Appendix A, “Cmdlet Naming Conventions.” In general, the cmdlets follow a standard naming convention such as Get-Help, Get-EventLog, or Get-Process. The “get” cmdlets display information about the item that is specified on the right side of the dash. The “set” cmdlets are used to modify or to set information about the item on the right side of the dash. An example of a “set” cmdlet is Set-Service, which can be used to change the startmode of a service. An explanation of this naming convention is found in Appendix A, “Cmdlet Naming Conventions.”

Configuring Windows PowerShell

Once Windows PowerShell is installed on a platform, there are still some configuration issues to address. This is in part due to the way the Windows PowerShell team at Microsoft perceives the use of the tool. For example, the Windows PowerShell team believes that 80 percent of Windows PowerShell users will not utilize the scripting features of Windows PowerShell; thus, the scripting capability is turned off by default. Find more information on enabling scripting support in Windows Power Shell in Chapter 2, “Scripting Windows PowerShell.”

Creating a Windows PowerShell Profile

There are many settings that can be stored in a Windows PowerShell profile. These items can be stored in a psconsole file. To export the console configuration file, use the Export-Console cmdlet as shown here:

```
PS C:\> Export-Console myconsole
```

The psconsole file is saved in the current directory by default, and will have an extension of .psc1. The psconsole file is saved in an .xml format; a generic console file is shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>1.0</PSVersion>
  <PSSnapIns />
</PSConsoleFile>
```

Configuring Windows PowerShell Startup Options

There are several methods available to start Windows PowerShell. For example, if the logo you receive when clicking the default Windows PowerShell icon seems to get in your way, you can launch without it. You can start Windows PowerShell using different profiles and even run a single Windows PowerShell command and exit the shell. If you need to start a specific version of Windows PowerShell, you can do that as well by supplying a value for the *version* parameter. Each of these options is illustrated in the following list.

- Launch Windows PowerShell without the banner by using the *-nologo* argument as shown here:

  ```
  PowerShell -nologo
  ```

- Launch a specific version of Windows PowerShell by using the *-version* argument:

  ```
  PowerShell -version 1.0
  ```

- Launch Windows PowerShell using a specific configuration file by specifying the *-psconsolefile* argument:

  ```
  PowerShell -psconsolefile myconsole.psc1
  ```

- Launch Windows PowerShell, execute a specific command, and then exit by using the *-command* argument. The command must be prefixed by the ampersand sign and enclosed in curly brackets:

  ```
  powershell -command "& {get-process}"
  ```

Security Issues with Windows PowerShell

As with any tool as versatile as Windows PowerShell, there are some security concerns. Security, however, was one of the design goals in the development of Windows PowerShell.

When you launch Windows PowerShell, it opens in your Users\userName folder; this ensures you are in a directory where you will have permission to perform certain actions and activities. This technique is far safer than opening at the root of the drive or opening in the system root.

To change to a directory, you can't automatically go up to the next level; you must explicitly name the destination of the change directory operation (but you can use the dotted notation with the Set-Location cmdlets as in Set-Location ..).

Running scripts is disabled by default but this can be easily managed with Group Policy or login scripts.

Controlling the Execution of Cmdlets

Have you ever opened a CMD interpreter prompt, typed in a command, and pressed Enter so you could see what happens? If that command happens to be Format C:\, are you sure you want to format your C drive? There are several arguments that can be passed to cmdlets to control the way they execute. These arguments will be examined in this section.

> **Tip** Most of the Windows PowerShell cmdlets support a "prototype" mode that can be entered by using the *-whatif* parameter. The implementation of the *whatif* switch can be decided by the person developing the cmdlet; however, the Windows PowerShell team recommends that developers implement *-whatif* if the cmdlet will make changes to the system.

Although not all cmdlets support these arguments, most of the cmdlets included with Windows PowerShell do. The three ways to control execution are *-whatif*, *-confirm*, and *suspend*. *Suspend* is not an argument that gets supplied to a cmdlet, but it is an action you can take at a confirmation prompt, and is therefore another method of controlling execution.

To use *-whatif,* first enter the cmdlet at a Windows PowerShell prompt. Then type the *-whatif* parameter after the cmdlet. The use of the *-whatif* argument is illustrated in the following WhatIf.txt example. On the first line, launch Notepad. This is as simple as typing the word **notepad** as shown in the path. Next, use the Get-Process cmdlet to search for all processes that begin with the name *note*. In this example, there are two processes with a name beginning with *notepad*. Next, use the Stop-Process cmdlet to stop a process with the name of *notepad,* but because the outcome is unknown, use the *-whatif* parameter. *Whatif* tells you that it will kill two processes, both of which are named *notepad*, and it also gives the process ID number so you can verify if this is the process you wish to kill. Just for fun, once again use the Stop-Process cmdlet to stop all processes with a name that begins with the letter *n*. Again, wisely use the *whatif* parameter to see what would happen if you execute the command.

WhatIf.txt

```
PS C:\Users\edwils> notepad
PS C:\Users\edwils> Get-Process note*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     45       2     1044       3904    53     0.03   3052 notepad
     45       2     1136       4020    54     0.05   3140 notepad


PS C:\Users\edwils> Stop-Process -processName notepad -WhatIf
What if: Performing operation "Stop-Process" on Target "notepad (3052)".
What if: Performing operation "Stop-Process" on Target "notepad (3140)".

PS C:\Users\edwils> Stop-Process -processName n* -WhatIf
What if: Performing operation "Stop-Process" on Target "notepad (3052)".
What if: Performing operation "Stop-Process" on Target "notepad (3140)".
```

So what happens if the *whatif* switch is not implemented? To illustrate this point, notice that in the following WhatIf2.txt example, when you use the New-Item cmdlet to create a new directory named myNewtest off the root, the *whatif* switch is implemented and it confirms that the command will indeed create C:\myNewtest.

Note what happens, however, when you try to use the *whatif* switch on the Get-Help cmdlet. You might guess it would display a message such as, "What if: Retrieving help information for

Get-Process cmdlet." But what is the point? As there is no danger with the Get-Help cmdlet, there is no need to implement *whatif* on Get-Help.

WhatIf2.txt

```
PS C:\Users\edwils> New-Item -Name myNewTest -Path c:\ -ItemType directory -WhatIf
What if: Performing operation "Create Directory" on Target
"Destination: C:\myNewTest".

PS C:\Users\edwils> get-help Get-Process -whatif
Get-Help : A parameter cannot be found that matches parameter name 'whatif'.
At line:1 char:28
+ get-help Get-Process -whatif <<<<
```

> **Best Practices** The use of the *-whatif* parameter should be considered an essential tool in the network administrator's repertoire. Using it to model commands before execution can save hours of work each year.

Confirming Commands

As you saw in the previous section, you can use *-whatif* to create a prototype cmdlet in Windows PowerShell. This is useful for checking what a command will do. However, to be prompted before the command executes, use the *-confirm* switch. In practice, using the *-confirm* switch can generally take the place of *-whatif*, as you will be prompted before the action occurs. This is shown in the ConfirmIt.txt example that follows.

In the ConfirmIt.txt file, first launch Calculator (Calc.exe). Because the file is in the path, you don't need to hard-code either the path or the extension. Next, use Get-Process with the c* wildcard pattern to find all processes that begin with the letter *c*. Notice that there are several process names on the list. The next step is to retrieve only the Calc.exe process. This returns a more manageable result set. Now use the Stop-Process cmdlet with the *-confirm* switch. The cmdlet returns the following information:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "calc (2924)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend
[?] Help (default is "Y"):
```

You will notice this information is essentially the same as the information provided by the *whatif* switch but it also provides the ability to perform the requested action. This can save time when executing a large number of commands.

ConfirmIt.txt

```
PS C:\Users\edwils> calc
PS C:\Users\edwils> Get-Process c*
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     43       2     1060       4212    54     0.03   2924 calc
   1408       7     3364       6556    81             372 casha
   1132      16    23156      34680   129            3084 CcmExec
    599       5     1680       4956    88             620 csrss
    480      10    15812      20500   195             688 csrss


PS C:\Users\edwils> Get-Process calc

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     43       2     1060       4212    54     0.03   2924 calc


PS C:\Users\edwils> Stop-Process -Name calc -Confirm

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "calc (2924)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?]
Help (default is "Y"): y
PS C:\Users\edwils> Get-Process c*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1412       7     3364       6556    81             372 casha
   1154      16    23224      34740   130            3084 CcmExec
    598       5     1680       4956    88             620 csrss
    477      10    15812      20488   195             688 csrss
```

Suspending Confirmation of Cmdlets

The ability to prompt for confirmation of a cmdlet's execution is extremely useful and at times may be vital in maintaining a high level of system uptime. For example, there are times when you have typed in a long command and then remember that you must perform another procedure first. In this case, simply suspend execution of the command. The commands used in the suspending execution of a cmdlet and associated output are shown in the following SuspendConfirmation.txt example.

In the SuspendConfirmation.txt file, first launch Microsoft Paint (Mspaint.exe). Because Mspaint.exe is in the path, you don't need to supply any path information to the file. You then get the process information by using the Get-Process cmdlet. Use the ms* wildcard, which matches any process name that begins with the letters *ms*. Once you have identified the correct process, use the Stop-Process cmdlet and the *confirm* switch. Instead of answering *yes* to the confirmation prompt, just suspend execution of the command so you can run an additional command (perhaps you forgot the process ID number). Once you have finished running the additional command, type **exit** to return to the suspended command from the nested prompt. Once you have killed the mspaint process, you can once again use the Get-Process cmdlet to confirm the process has been killed.

SuspendConfirmation.txt

```
PS C:\Users\edwils> mspaint
PS C:\Users\edwils> Get-Process ms*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     98       4     5404      10492    72     0.09   3064 mspaint


PS C:\Users\edwils> Stop-Process -id 3064 -Confirm

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "mspaint (3064)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): s
PS C:\Users\edwils>>> Get-Process ms*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     97       4     5404      10496    72     0.09   3064 mspaint


PS C:\Users\edwils>>> exit

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "mspaint (3064)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
PS C:\Users\edwils> Get-Process ms*
```

Supplying Options for Cmdlets

As you have seen in the previous sections, you can use *-whatif* and *-confirm* to control the execution of cmdlets. One question students often ask me is, "How do I know what options are available?" The answer is that the Windows PowerShell team created a set of standard options. These standard options are called *common parameters*. When you look at the syntax description for a cmdlet, often it will state that the cmdlet supports the common parameters. This is shown here for the Get-Process cmdlet:

```
SYNTAX
    Get-Process [[-name] <string[]>] [<CommonParameters>]

    Get-Process -id <Int32[]> [<CommonParameters>]

    Get-Process -inputObject <Process[]> [<CommonParameters>]
```

One of the useful features of Windows PowerShell is the standardization of the syntax in working with cmdlets. This vastly simplifies learning the new shell and language. Table 1-1 lists the common parameters. Keep in mind that all cmdlets will not implement all of these parameters. However, if the parameters are used they will be interpreted in the same way for all cmdlets because the Windows PowerShell engine interprets the parameters.

Table 1-1 Common Parameters

Parameter	Meaning
-whatif	Tells the cmdlet not to execute; instead it will tell you what would happen if the cmdlet were to actually run.
-confirm	Tells the cmdlet to prompt prior to executing the command.
-verbose	Instructs the cmdlet to provide a higher level of detail than a cmdlet not using the verbose parameter.
-debug	Instructs the cmdlet to provide debugging information.
-erroraction	Instructs the cmdlet to perform a certain action when an error occurs. Allowable actions are: continue, stop, SilentlyContinue, and inquire.
-errorvariable	Instructs the cmdlet to use a specific variable to hold error information. This is in addition to the standard *$error* variable.
-outvariable	Instructs the cmdlet to use a specific variable to hold the output information.
-outbuffer	Instructs the cmdlet to hold a certain number of objects prior to calling the next cmdlet in the pipeline.

Working with Get-Help

Windows PowerShell is intuitively easy to use; learn simply by doing. Online help makes it even easier to use the program. The help system in Windows PowerShell can be entered by several methods. To learn about using Windows PowerShell, use the Get-Help cmdlet as shown here:

```
get-help get-help
```

This command prints out help about the Get-Help cmdlet. The output from this cmdlet is shown here:

```
NAME
    Get-Help

SYNOPSIS
    Displays information about Windows PowerShell cmdlets and concepts.


SYNTAX
    Get-Help [[-name] <string>] [-component <string[]>] [-functionality <string
    []>] [-role <string[]>] [-category <string[]>] [-full] [<CommonParameters>]

    Get-Help [[-name] <string>] [-component <string[]>] [-functionality <string
    []>] [-role <string[]>] [-category <string[]>] [-detailed] [<CommonParamete
    rs>]

    Get-Help [[-name] <string>] [-component <string[]>] [-functionality <string
    []>] [-role <string[]>] [-category <string[]>] [-examples] [<CommonParamete
    rs>]
```

```
    Get-Help [[-name] <string>] [-component <string[]>] [-functionality <string
    []>] [-role <string[]>] [-category <string[]>] [-parameter <string>] [<Comm
    onParameters>]


DETAILED DESCRIPTION
    The Get-Help cmdlet displays information about Windows PowerShell cmdlets
    and concepts. You can also use "Help {<cmdlet name> | <topic-name>" or "<cmd
    let-name> /?". "Help" displays the help topics one page at a time. The "/?"
     displays help for cmdlets on a single page.


RELATED LINKS
    Get-Command
    Get-PSDrive
    Get-Member

REMARKS
    For more information, type: "get-help Get-Help -detailed".
    For technical information, type: "get-help Get-Help -full".
```

The awesome thing about online help for Windows PowerShell, is that not only does it display help about commands—which you would expect—but it also has three different levels of display: *normal*, *detailed*, and *full*. Additionally, you can obtain help about concepts in Windows PowerShell. This last feature is equivalent to having an online instruction manual. To retrieve a listing of all the conceptual help articles, use the Get-Help about* command as shown here:

```
get-help about*
```

Suppose you do not remember the exact name of the cmdlet you wish to use but you remember it was a “get” cmdlet. You can use a wildcard (such as *) to obtain the name of the cmdlet. This is shown here:

```
get-help get*
```

This technique of using a wildcard operator can be extended further. If you remember the cmdlet was a “get” cmdlet and it started with the letter *p* you could use the following syntax to retrieve the desired cmdlet:

```
get-help get-p*
```

Suppose, however, that you know the exact name of the cmdlet but you can’t exactly remember the syntax. For this scenario, you could use the *-examples* argument. To retrieve several examples of the Get-PSDrive cmdlet, you could use Get-Help with the *-examples* argument as shown here:

```
get-help get-psdrive -examples
```

To see help displayed one page at a time, you can use the help function which displays the help output text through the *more* function. This is useful if you want to avoid scrolling up and down to see the help output. This command is shown here:

```
get-help get-help | more
```

The formatted output from the *more* function is shown in Figure 1-3.

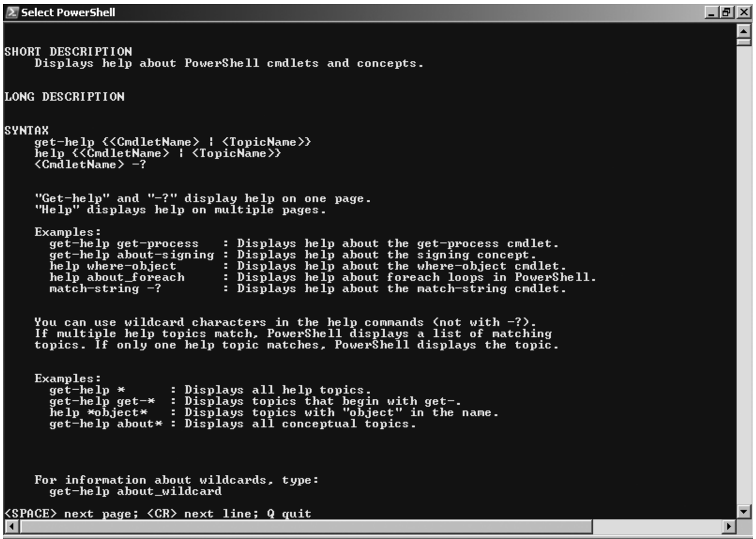

Figure 1-3 By using the *more* function, you can display lengthy help topics one page at a time.

To obtain detailed help about the Get-Help cmdlet, use the *-detailed* argument as shown here:

```
get-help get-help -detailed
```

If you want to retrieve technical information about the Get-Help cmdlet, use the *-full* argument. This is shown here:

```
get-help get-help -full
```

Getting tired of typing Get-Help over and over? After all, it is eight characters long and one of them is a dash. The solution is to create an alias to the Get-Help cmdlet. An alias is a shortcut keystroke combination that will launch a program or cmdlet when typed. In the create Get-Help alias for this example, you can assign the Get-Help to the *gh* key combination.

> **Tip** Before creating an alias for a cmdlet, confirm there is not already an alias to the cmdlet by using Get-Alias. Then use Set-Alias to assign the cmdlet to a unique keystroke combination.

Working with Aliases to Assign Shortcut Names to Cmdlets

Aliases allow you to assign shortcut names to cmdlets. This can greatly simplify working at the Windows PowerShell prompt and it will allow you to customize the command syntax as you prefer. As an example, suppose you want to create an alias for the Get-Help cmdlet. Instead of typing Get-Help, perhaps you prefer to type *gh*. This can be accomplished in four simple steps. First, ensure there is not already an alias assigned to the desired keystroke combination to avoid confusion. The next thing you might want to do is review help for the Set-Alias cmdlet. Once you have done this, call the Set-Alias cmdlet and pass the new name you want to create and the name of the cmdlet you wish to alias. After you have created the alias, you may want to use Get-Alias to verify the alias was created properly. The completed code from this section is in the GhAlias.txt file in the chapter01 folder on the companion CD-ROM.

1. Retrieve an alphabetic listing of all currently defined aliases and inspect the list for one assigned to either the Get-Help cmdlet or for the keystroke combination *gh*. The command to do this is shown here:

   ```
   get-alias |sort
   ```

2. Once you have determined there is no alias for the Get-Help cmdlet and that none is assigned to the *gh* keystroke combination, review the syntax for the Set-Alias cmdlet. Use the *-full* argument to the Get-Help cmdlet. This is shown here:

   ```
   get-help set-alias -full
   ```

3. Use the Set-Alias cmdlet to assign the *gh* keystroke combination to the Get-Help cmdlet. To do this, use the following command:

   ```
   set-alias gh get-help
   ```

4. Use the Get-Alias cmdlet to verify the alias was properly created. To do this, use the following command:

   ```
   Get-Alias gh
   ```

> **Tip** If the syntax of Set-Alias is a little confusing, you can use named parameters instead of the default positional binding. In addition, I recommend using either the *whatif* switch or the *confirm* switch. You can also specify a description for the alias. The modified syntax would look like this:
>
> ```
> Set-Alias -Name gh -Value Get-Help -Description "mred help alias" -WhatIf
> ```

As you have seen, Windows PowerShell can be used as a replacement to the CMD interpreter. But it also has a large number of built-in cmdlets that provide the opportunity to perform a plethora of activities. These cmdlets can be used either in a stand-alone fashion or they can be run together as a group.

Accessing Windows PowerShell

Once Windows PowerShell is installed, it immediately becomes available for use. However, pressing R while pressing the Windows flag key on your keyboard to bring up the Windows Run dialog box or mousing around—doing the old Start button/Run dialog box thing and typing PowerShell all the time—becomes somewhat less helpful. I created a shortcut to Windows PowerShell and placed that shortcut on my desktop. For me and the way I work, this is ideal. This is so useful, in fact, that I wrote a script to perform this function. This script can be called via a logon script, to automatically create the shortcut on the desktop. The script is named CreateShortCutToPowerShell.vbs:

CreateShortCutToPowerShell.vbs

```
Option Explicit
Dim objshell
Dim strDesktop
Dim objshortcut
Dim strProg
strProg = "powershell.exe"

Set objshell=CreateObject("WScript.Shell")
strDesktop = objshell.SpecialFolders("desktop")
set objShortcut = objshell.CreateShortcut(strDesktop & "\powershell.lnk")
objshortcut.TargetPath = strProg
objshortcut.WindowStyle = 1
objshortcut.Description = funfix(strProg)
objshortcut.WorkingDirectory = "C:\"
objshortcut.IconLocation= strProg
objshortcut.Hotkey = "CTRL+SHIFT+P"
objshortcut.Save

Function funfix(strin)
funfix = InStrRev(strin,".")
funfix = Mid(strin,1,funfix)
End function
```

Additional Uses of Cmdlets

Now that you have learned about using the help utilities and working with aliases, it's time to examine some additional ways to use cmdlets in Windows PowerShell.

Tip To save time when typing the cmdlet name, simply type enough of the cmdlet name to uniquely distinguish it, and then press the Tab key. What is the result? Tab completion finishes the cmdlet name for you. This also works with argument names and other procedures. Feel free to experiment with this great timesaving technique. You may never have to type **get-command** again!

As the cmdlets return objects instead of "string values" you can obtain additional information about the returned objects. This additional information would not be available if you were working with just string data. To obtain additional information, use the pipe character (|), then take information from one cmdlet and feed it to another cmdlet. This may seem complicated, but in reality, it is quite simple. By the end of this chapter, the procedure should seem quite natural.

At the most basic level, consider the simple example of obtaining and formatting a directory listing. After you retrieve the directory listing, you may want to format the way it is displayed, perhaps as either a table or a list. As you can see, there are two separate operations: obtaining the directory listing and formatting the list. This formatting task takes place on the right side of the pipe after the directory listing has been gathered. This is the way pipelines work. Now, let's examine them in action while looking at the Get-ChildItem cmdlet.

Using the Get-ChildItem Cmdlet

Earlier in this chapter, you used the *dir* command to obtain a listing of all the files in a directory. This works because there is an alias built into Windows PowerShell that assigns the Get-ChildItem cmdlet to the letter combination *dir*. We can verify this by using the Get-Alias cmdlet. This is shown in the GetDirAlias.txt file.

GetDirAlias.txt

```
PS C:\> Get-Alias dir

CommandType     Name       Definition
-----------     ----       ----------
Alias           dir        Get-ChildItem
```

In Windows PowerShell, there really is no cmdlet named *dir*, nor does it actually use the *dir* command. The alias *dir* is associated with the Get-ChildItem cmdlet. This is why the output from *dir* is different in Windows PowerShell than it is in the Cmd.exe interpreter. The alias *dir* is shown here when you use the Get-Alias cmdlet to resolve the association.

> **Tip** When using Get-ChildItem to produce a directory listing, use the *force* switch if you want to view hidden and system files and folders. It would look like this: Get-ChildItem *-Force*.

Formatting Output

There are four format cmdlets included with Windows PowerShell. Of these cmdlets, you will routinely use three: Format-List, Format-Wide, and Format-Table. The fourth cmdlet, Format-Custom, can display output in a fashion that is not a list, table, or wide format. It accomplishes this by using a *.format.ps1xml file. You can use either the default view contained in the *.format.ps1xml files or you can define your own format.ps1xml file.

Let's look at formatting output utilizing the remaining three format cmdlets beginning with the most useful of the three: Format-List.

Format-List

Format-List is one of the core cmdlets you will use time and again. For example, if you use the Get-WmiObject cmdlet to look at the properties of the *Win32_LogicalDisk* class, you will receive a minimum listing of the default properties of the class. This listing is shown here:

```
PS C:\> Get-WmiObject Win32_LogicalDisk


DeviceID     : C:
DriveType    : 3
ProviderName :
FreeSpace    : 10559041536
Size         : 78452355072
VolumeName   : Sea Drive
```

Although in many cases this behavior is fine, there are times when you may be interested in the other properties of the class. The first thing to do when exploring other properties that may be available is to use the wildcard *. This will list all the properties as shown here:

```
PS C:\> Get-WmiObject Win32_LogicalDisk | Format-List *


Status                       :
Availability                 :
DeviceID                     : C:
StatusInfo                   :
__GENUS                      : 2
__CLASS                      : Win32_LogicalDisk
__SUPERCLASS                 : CIM_LogicalDisk
__DYNASTY                    : CIM_ManagedSystemElement
__RELPATH                    : Win32_LogicalDisk.DeviceID="C:"
__PROPERTY_COUNT             : 40
__DERIVATION                 : {CIM_LogicalDisk, CIM_StorageExtent,
CIM_LogicalDevice, CIM_LogicalElement...}
__SERVER                     : M5-1875135
__NAMESPACE                  : root\cimv2
__PATH                       : \\M5-1875135\root\cimv2:Win32_LogicalDisk.DeviceID="C:"
Access                       : 0
BlockSize                    :
Caption                      : C:
Compressed                   : False
ConfigManagerErrorCode       :
ConfigManagerUserConfig      :
CreationClassName            : Win32_LogicalDisk
Description                  : Local Fixed Disk
DriveType                    : 3
ErrorCleared                 :
ErrorDescription             :
ErrorMethodology             :
FileSystem                   : NTFS
```

```
FreeSpace                    : 10559041536
InstallDate                  :
LastErrorCode                :
MaximumComponentLength       : 255
MediaType                    : 12
Name                         : C:
NumberOfBlocks               :
PNPDeviceID                  :
PowerManagementCapabilities  :
PowerManagementSupported     :
ProviderName                 :
Purpose                      :
QuotasDisabled               :
QuotasIncomplete             :
QuotasRebuilding             :
Size                         : 78452355072
SupportsDiskQuotas           : False
SupportsFileBasedCompression : True
SystemCreationClassName      : Win32_ComputerSystem
SystemName                   : M5-1875135
VolumeDirty                  :
VolumeName                   : Sea Drive
VolumeSerialNumber           : F0FE15F7
```

Once you have looked at all the properties that are available for a particular class, you can then choose only the properties you are interested in. Replace the wildcard * with the property names gleaned from the preceding listing. This technique is shown here:

```
PS C:\> Get-WmiObject Win32_LogicalDisk | Format-List Name, FileSystem, FreeSpace


Name       : C:
FileSystem : NTFS
FreeSpace  : 10559029248
```

Instead of typing a long list of property names, you can choose a range of property names by using wildcard characters. To see only the property names that begin with the letter *f*, you can use the technique shown here:

```
PS C:\> Get-WmiObject Win32_LogicalDisk | Format-List f*


FileSystem : NTFS
FreeSpace  : 10558660608
```

If you want to see properties that begin with *n* and with *f*, then you need to introduce square brackets as shown here:

```
PS C:\> Get-WmiObject Win32_LogicalDisk | Format-List [nf]*


FileSystem     : NTFS
FreeSpace      : 10558238720
Name           : C:
NumberOfBlocks :
```

These commands, with their associated complete output, can be found in the Format-List.txt file in the chapter01 folder on the companion CD-ROM.

Format-Table

The Format-Table cmdlet provides a number of features that make it especially well suited for network management tasks. In particular, it produces columns of data that allow for quick viewing. As with Format-List and Format-Wide, you can choose the properties you wish to display, and in so doing, easily eliminate distracting data from annoyingly verbose cmdlets. In the example shown here, first take a recursive look through the hard drive to find all the log files (those designated with the .log extension). While the output is considerable, it has been trimmed here to show a sample of the output. The Format-Table cmdlet is used to produce the output from the Get-ChildItem cmdlet shown here:

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log | Format-Table


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Backup_Extras_92705


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          8/3/2004   6:34 PM    3931872 setupapi.log
-a---          8/2/2004   9:32 PM     206168 Windows Update.log
-a---          6/8/2004  12:41 AM     170095 wmsetup.log
```

In addition to relying on the default behavior of the cmdlet, you can also choose specific properties. One issue with this approach, as shown here, is that the formatting uses the existing screen resolution for the window, thus you often end up with columns on opposite sides of the window. This can be acceptable for a quick-and-dirty column list, but it is not a format for saving data.

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log | Format-Table
-Property name, length, lastWriteTime

Name                                                                      Length
LastWriteTime
----
------ -------------
setupapi.log                                                             3931872
8/3/2004 6:34:53 PM
Windows Update.log                                                        206168
8/2/2004 9:32:06 PM
wmsetup.log                                                               170095
6/8/2004 12:41:32 AM
Debug.log                                                                      0
8/23/2006 8:10:38 PM
AVCheck.Log                                                               191694
5/8/2007 9:28:05 AM
AVCheckServer.Log                                                           7762
5/8/2007 9:28:05 AM
```

To produce a list that uses the window size a bit more efficiently, you can specify the *autosize* switch. There is only one thing to keep in mind when using the *autosize* switch: It needs to know the length of the longest item to be stored in each column. To do this, the switch must wait until all objects have been enumerated, then it will determine the maximum length of each column and determine the size of the listing. This can cause the command execution to block until all items have enumerated, so this process takes a while to complete. You may not want to wait for the *autosize* to enumerate a large collection of objects if you are in a hurry, for example, working on a server-down issue. For small object sets, the performance hit is negligible; however, with a command that takes a long time to complete, such as this one, the difference is noticeable. The difference in output, however, is also noticeable (and you will probably feel it is worth the wait to have a more manageable output).

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log | Format-Table
-Property name, length, lastWriteTime -AutoSize

Name                                                    Length LastWriteTime
----                                                    ------ -------------
setupapi.log                                           3931872 8/3/2004 6:34:53 PM
Windows Update.log                                      206168 8/2/2004 9:32:06 PM
wmsetup.log                                             170095 6/8/2004 12:41:32 AM
Debug.log                                                    0 8/23/2006 8:10:38 PM
AVCheck.Log                                             191694 5/8/2007 9:28:05 AM
```

The last thing to look at in conjunction with Format-Table is pairing it with the Sort-Object cmdlet. Sort-Object allows you to organize data by property and to display it in a sorted fashion. In this example, the alias for Sort-Object (sort) is used, which reduces the amount of typing necessary. The command is still rather long and is wrapped here for readability. (To be honest, when commands begin to reach this length, I have a tendency to turn the process into a script.) When you examine the following command, notice that the data is sorted before feeding it to the Format-Table cmdlet. Please note that by default the Sort-Object cmdlet sorts in ascending (smallest to largest) order. If desired, you can specify the *-descending* switch to see the files organized from largest to smallest.

```
PS C:\>Get-ChildItem c:\ -Recurse -Include *.log | Sort -Property
length | Format-Table name, lastwriteTime, length -AutoSize
Name                                                  LastWriteTime          Length
----                                                  -------------          ------
PASSWD.LOG                                            5/10/2007 2:44:58 AM        0
sam.log                                               11/29/2006 1:14:33 PM       0
poqexec.log                                           2/1/2007 6:50:49 PM         0
ChkAcc.log                                            5/10/2007 2:45:00 AM        0
Debug.log                                             8/23/2006 8:10:38 PM        0
setuperr.log                                          3/16/2007 7:18:17 AM        0
setuperr.log                                          4/4/2007 6:34:54 PM         0
netlogon.log                                          2/1/2007 7:04:44 PM         3
```

There are also other ways to sort. For example, you can sort the list of log files by date modified in descending order. By doing this, you can see the most recently modified log files. To perform this procedure, you need to modify the sort object. The remainder of the command is the same. A portion of this output is shown here. It is interesting to note that the majority of these logs were modified during the log-on process.

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log | Sort -Property
lastWriteTime -descending | Format-Table name, lastwriteTime, length -AutoSize
Name                                                    LastWriteTime          Length
----                                                    -------------          ------
mtrmgr.log                                              5/10/2007 4:56:52 AM   1538364
LocationServices.log                                    5/10/2007 4:56:26 AM    830557
StateMessage.log                                        5/10/2007 4:55:00 AM    129595
Scheduler.log                                           5/10/2007 4:55:00 AM    393352
StatusAgent.log                                         5/10/2007 4:53:24 AM    723564
edb.log                                                 5/10/2007 4:51:49 AM    131072
PolicyEvaluator.log                                     5/10/2007 4:51:25 AM   1672613
ClientLocation.log                                      5/10/2007 4:51:24 AM    330046
FSPStateMessage.log                                     5/10/2007 4:51:18 AM    228879
CBS.log                                                 5/10/2007 4:46:55 AM  28940091
CertificateMaintenance.log                              5/10/2007 4:42:17 AM    206472
CcmExec.log                                             5/10/2007 4:00:51 AM    537177
wmiprov.log                                             5/10/2007 3:03:11 AM     19503
PolicyAgentProvider.log                                 5/10/2007 2:54:02 AM    252866
UpdatesHandler.log                                      5/10/2007 2:53:19 AM    108552
CIAgent.log                                             5/10/2007 2:53:19 AM     99114
ScanAgent.log                                           5/10/2007 2:53:18 AM    354939
UpdatesDeployment.log                                   5/10/2007 2:53:18 AM   1106297
SrcUpdateMgr.log                                        5/10/2007 2:53:02 AM    151452
smssha.log                                              5/10/2007 2:52:02 AM    107104
execmgr.log                                             5/10/2007 2:52:02 AM    150942
InventoryAgent.log                                      5/10/2007 2:52:02 AM     34034
ServiceWindowManager.log                                5/10/2007 2:52:02 AM    139955
SdmAgent.log                                            5/10/2007 2:49:46 AM    172101
UpdatesStore.log                                        5/10/2007 2:49:43 AM     64787
WUAHandler.log                                          5/10/2007 2:49:39 AM     14590
CAS.log                                                 5/10/2007 2:49:35 AM    198955
PeerDPAgent.log                                         5/10/2007 2:49:35 AM      7900
PolicyAgent.log                                         5/10/2007 2:49:35 AM    246873
RebootCoordinator.log                                   5/10/2007 2:49:35 AM     20420
InternetProxy.log                                       5/10/2007 2:49:34 AM     85825
ClientIDManagerStartup.log                              5/10/2007 2:49:34 AM    158351
WindowsUpdate.log                                       5/10/2007 2:46:46 AM   1553462
edb.log                                                 5/10/2007 2:46:43 AM     65536
setupapi.dev.log                                        5/10/2007 2:46:38 AM   6469237
setupapi.app.log                                        5/10/2007 2:46:38 AM   2722285
WMITracing.log                                          5/10/2007 2:45:57 AM  16777216
ChkAcc.log                                              5/10/2007 2:45:00 AM         0
PASSWD.LOG                                              5/10/2007 2:44:58 AM         0
```

If you look at the Format-Table.txt file in the chapter01 folder, you will notice there are many errors in the log file. This is because the Get-ChildItem cmdlet attempted to access directories and files that are protected, causing access-denied messages. During development these errors are helpful to let you know that you are not accessing files and folders; however, they become problematic once you begin to analyze the data. An example of one of these errors is shown here:

```
Get-ChildItem : Access to the path 'C:\Windows\CSC' is denied.
At line:1 char:14
```

The error message is helpful in that it tells you the name of the cmdlet that caused the error and the action that provoked the error. You can eliminate these types of errors by using the *-ErrorAction* common parameter on the Get-ChildItem cmdlet, specifying the SilentlyContinue keyword. This modified line of code is shown here:

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log -errorAction SilentlyContinue
| Sort -Property lastWriteTime -descending | Format-Table name, lastwriteTime,
length -AutoSize
```

Format-Wide

The Format-Wide cmdlet is not nearly as useful as Format-Table or Format-List. This is due to the limitation of displaying only one property per object. It can be useful, however, to have such a list. For example, suppose you only want a list of the processes running on your computer. You can use Get-Process cmdlet, and pipeline the resulting object to the Format-Wide cmdlet. This is shown here:

```
PS C:\> Get-Process | Format-Wide


ApMsgFwd                                                    ApntEx
Apoint                                                      audiodg
casha                                                       CcmExec
csrss                                                       csrss
dwm                                                         explorer
FwcAgent                                                    Idle
InoRpc                                                      InoRT
InoTask                                                     lsass
lsm                                                         mobsync
MSASCui                                                     powershell
powershell                                                  PowerShellIDE
rundll32                                                    SearchFilterHost
SearchIndexer                                               SearchProtocolHost
services                                                    SLsvc
smss                                                        spoolsv
SRUserService                                               svchost
svchost                                                     svchost
svchost                                                     svchost
svchost                                                     svchost
svchost                                                     svchost
svchost                                                     svchost
svchost                                                     svchost
svchost                                                     svchost
System                                                      taskeng
taskeng                                                     ThpSrv
```

```
ThpSrv                                                  TODDSrv
wininit                                                 winlogon
WINWORD                                                 wmdc
WmiPrvSE                                                WmiPrvSE
```

The output, while serviceable, uses a lot of lines on the console and it also wastes quite a bit of screen real estate. A better output can be obtained by using the *-column* parameter. This is illustrated here:

```
PS C:\> Get-Process | Format-Wide -Column 4
```

Although the four-column output cuts the list length by half, it still does not maximize all the available screen space. Though it might be possible to write a script that will figure out the optimum value of the *-column* parameter, such as the following DemoFormatWide.ps1 script, it is hardly worth the time and the trouble to pursue such an undertaking.

DemoFormatWide.ps1

```
function funGetProcess()
{
 if ($args)
  {
   Get-Process |
   Format-Wide -autosize
  }
 else
  {
   Get-Process |
   Format-Wide -column $i
  }
}

cls
$i = 1
for
   ($i ; $i -le 10 ; $i++)
{
  Write-Host -ForegroundColor red "`$i is equal to $i"
  funGetProcess
}
  Write-Host -ForeGroundColor red "Now use format-wide -autosize"
  funGetProcess("auto")
```

A better option for finding the optimum screen configuration for Format-Wide is to use the *-autosize* switch, shown here:

```
PS C:\> Get-Process | Format-Wide -AutoSize
```

Using the Get-Command Cmdlet

There are three cmdlets that are analogous to the three key spices used in Cajun cooking. You can make anything in the Cajun style of cooking if you remember: salt, pepper, and paprika. You want to make Cajun green beans? Add some salt, pepper, and paprika. You want to work with Windows PowerShell? Remember the "Cajun" cmdlets: Get-Help, Get-Command, and Get-Member. Calling on these three cmdlets, you can master Windows PowerShell. Since you have already looked at Get-Help, the next cmdlet to examine is Get-Command.

The most basic use of Get-Command is to produce a listing of commands available to Windows PowerShell. This is useful if you want to quickly see which cmdlets are available. This elementary use of Get-Command is illustrated here. One point to notice is that the definition is truncated.

```
PS C:\> Get-Command

CommandType     Name                                                Definition
-----------     ----                                                ----------
Cmdlet          Add-Content                                         Add-Content
[-Path] <String[]> [-Value] <Object[...
Cmdlet          Add-History                                         Add-History
[[-InputObject] <PSObject[]>] [-Pass...
Cmdlet          Add-Member                                          Add-Member
[-MemberType] <PSMemberTypes> [-Name]...
Cmdlet          Add-PSSnapin                                        Add-PSSnapin
[-Name] <String[]> [-PassThru] [-Ve...
Cmdlet          Clear-Content                                       Clear-Content
[-Path] <String[]> [-Filter <Strin...
Cmdlet          Clear-Item                                          Clear-Item
[-Path] <String[]> [-Force] [-Filter ...
```

By default, Get-Command is limited to producing a listing of cmdlets; therefore the cmdlet field is redundant. A nicer format of the list can be achieved by pipelining the resulting object into the Format-List cmdlet and choosing only the name and definition. This is illustrated here. As you can see in the code, this output is much easier to read and it provides the syntactical definition of each command:

```
PS C:\> Get-Command | Format-List name, definition


Name       : Add-Content
Definition : Add-Content [-Path] <String[]> [-Value] <Object[]> [-PassThru]
[-Filter <String>] [-Include <String[]>] [-Exclude <String[]>] [-Force]
[-Credential<PSCredential>] [-Verbose] [-Debug] [-ErrorAction <ActionPreference>]
[-ErrorVariable<String>] [-OutVariable <String>] [-OutBuffer <Int32>] [-WhatIf]
[-Confirm][-Encoding <FileSystemCmdletProviderEncoding>]            Add-Content
[-LiteralPath] <String[]> [-Value] <Object[]> [-PassThru][-Filter <String>]
[-Include <String[]>] [-Exclude <String[]>] [-Force] [-Credential<PSCredential>]
[-Verbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable
<String>] [-OutVariable <String>] [-OutBuffer <Int32>] [-WhatIf] [-Confirm]
[-Encoding <FileSystemCmdletProviderEncoding>]


Name       : Add-History
Definition : Add-History [[-InputObject] <PSObject[]>] [-Passthru] [-Verbose]
[-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable
String>] [-OutBuffer <Int32>]
```

So far, we have looked at normal usage of the Get-Command cmdlet. However, a more interesting method uses our knowledge of the noun and verb combination of cmdlet names. Armed with this information, we can look for commands that have a noun-called process in the name of the cmdlet This command would look like the following:

```
PS C:\> Get-Command -Noun process

CommandType     Name                                                             Definition
-----------     ----                                                             ----------
Cmdlet          Get-Process                                                      Get-Process
[[-Name] <String[]>] [-Verbose] [-De...
Cmdlet          Stop-Process                                                     Stop-Process
[-Id] <Int32[]> [-PassThru] [-Verbo...
```

Using this procedure, if you want to find a cmdlet that contains the letter *p* in the noun portion of the name, you can use wildcards to assist. This can reduce typing and help you explore available cmdlets. This command is shown here:

```
PS C:\> get-command -Noun p*

CommandType     Name                                                             Definition
-----------     ----                                                             ----------
Cmdlet          Add-PSSnapin                                                     Add-PSSnapin
[-Name] <String[]> [-PassThru] [-Ve...
Cmdlet          Convert-Path                                                     Convert-Path
[-Path] <String[]> [-Verbose] [-Deb...
Cmdlet          Get-PfxCertificate                                               Get-PfxCertificate [-
FilePath] <String[]> [-Verb...
Cmdlet          Get-Process                                                      Get-Process
[[-Name] <String[]>] [-Verbose] [-De...
Cmdlet          Get-PSDrive                                                      Get-PSDrive
[[-Name] <String[]>] [-Scope <String...
Cmdlet          Get-PSProvider                                                   Get-PSProvider
[[-PSProvider] <String[]>] [-Verb...
Cmdlet          Get-PSSnapin                                                     Get-PSSnapin
[[-Name] <String[]>] [-Registered] ...
Cmdlet          Join-Path                                                        Join-Path
[-Path] <String[]> [-ChildPath] <Strin...
Cmdlet          New-PSDrive                                                      New-PSDrive
[-Name] <String> [-PSProvider] <Stri...
Cmdlet          Out-Printer                                                      Out-Printer
[[-Name] <String>] [-InputObject <PS...
Cmdlet          Remove-PSDrive                                                   Remove-PSDrive
[-Name] <String[]> [-PSProvider <...
Cmdlet          Remove-PSSnapin                                                  Remove-PSSnapin
[-Name] <String[]> [-PassThru] [...
Cmdlet          Resolve-Path                                                     Resolve-Path
[-Path] <String[]> [-Credential <PS...
Cmdlet          Set-PSDebug                                                      Set-PSDebug
[-Trace <Int32>] [-Step] [-Strict] [...
Cmdlet          Split-Path                                                       Split-Path
[-Path] <String[]> [-LiteralPath <Str...
Cmdlet          Stop-Process                                                     Stop-Process
```

```
[-Id] <Int32[]> [-PassThru] [-Verbo...
Cmdlet          Test-Path                                               Test-Path
[-Path] <String[]> [-Filter <String>] ...
Cmdlet          Write-Progress                                          Write-Progress
[-Activity] <String> [-Status] <S...
```

By default, the Get-Command cmdlet displays only cmdlets; however, it can retrieve other items as well—even .exe files and .dll files. This is because Get-Command will display information about every item you can run in Windows PowerShell. An example of this is shown here in a listing of commands that contains the word *file* in the name. One point to remember: Only Windows PowerShell entities are displayed.

```
PS C:\> get-command -Name *file*

CommandType     Name                                                    Definition
-----------     ----                                                    ----------
Application     avifile.dll
C:\Windows\system32\avifile.dll
Application     filemgmt.dll
C:\Windows\system32\filemgmt.dll
Application     FileSystem.format.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\FileS...
Application     filetrace.mof
C:\Windows\System32\Wbem\filetrace.mof
Application     forfiles.exe
C:\Windows\system32\forfiles.exe
```

You can easily correct this behavior by using the *-commandType* parameter and limiting the search to cmdlets. This modified command is shown here:

```
PS C:\> get-command -Name *file* -CommandType cmdlet

CommandType     Name                                                    Definition
-----------     ----                                                    ----------
Cmdlet          Out-File                                                Out-File
[-FilePath] <String> [[-Encoding] <Stri
```

These examples give you an idea of the types of searches you can perform with the Get-Command cmdlet. These commands and their associated output are contained in the Get-Command.txt file in the chapter01 folder on the companion CD-ROM.

Exploring with the Get-Member Cmdlet

The third important cmdlet provided with Windows PowerShell is Get-Member. Some students look askance when I introduce Get-Member as one of the three “Cajun” cmdlets. Indeed, I had one student who raised his hand and asked what it was good for. This is a fair question. The thing that makes Get-Member so useful is that it can tell you which properties and methods are supported by an object. If you remember that everything in Windows PowerShell is an object, then you are well on your way to achieving enlightenment with this command. Perhaps a simple example will illustrate the value of this cmdlet.

If you have a folder named mytest, and use the Get-Item cmdlet to obtain an object that represents the folder, you can store this reference in a variable named *$a*. This is shown here:

```
PS C:\> $a = Get-Item c:\mytest
```

Once you have an instance of the folder object contained in the *$a* variable, you can examine the methods and properties of a folder object by pipelining the object into the Get-Member cmdlet. This command and associated output are shown here:

```
PS C:\> $a | Get-Member


   TypeName: System.IO.DirectoryInfo

Name                       MemberType     Definition
----                       ----------     ----------
Create                     Method         System.Void Create(), System.Void
Create(DirectorySecurity directorySecurity)
CreateObjRef               Method         System.Runtime.Remoting.ObjRef
CreateObjRef(Type requestedType)
CreateSubdirectory         Method         System.IO.DirectoryInfo
CreateSubdirectory(String path), System.IO.Director...
Delete                     Method         System.Void Delete(), System.Void
Delete(Boolean recursive)
Equals                     Method         System.Boolean Equals(Object obj)
GetAccessControl           Method         System.Security.AccessControl.DirectorySecurity Get
AccessControl(), System
GetDirectories             Method         System.IO.DirectoryInfo[]
GetDirectories(), System.IO.DirectoryInfo[GetFiles                  Method         System.IO
.FileInfo[] GetFiles(String searchPattern), System.IO.FileInfo[] G...
GetFileSystemInfos        Method         System.IO.FileSystemInfo[] GetFileSystemInfos(String
searchPattern), System...
GetHashCode                Method         System.Int32 GetHashCode()
GetLifetimeService         Method         System.Object GetLifetimeService()
GetObjectData              Method         System.Void GetObjectData
*(SerializationInfo info, StreamingContext context)
GetType                    Method         System.Type GetType()
get_Attributes             Method         System.IO.FileAttributes get_Attributes()
get_CreationTime           Method         System.DateTime get_CreationTime()
get_CreationTimeUtc        Method         System.DateTime get_CreationTimeUtc()
get_Exists                 Method         System.Boolean get_Exists()
get_Extension              Method         System.String get_Extension()
get_FullName               Method         System.String get_FullName()
get_LastAccessTime         Method         System.DateTime get_LastAccessTime()
get_LastAccessTimeUtc      Method         System.DateTime get_LastAccessTimeUtc()
get_LastWriteTime          Method         System.DateTime get_LastWriteTime()
get_LastWriteTimeUtc       Method         System.DateTime get_LastWriteTimeUtc()
get_Name                   Method         System.String get_Name()
get_Parent                 Method         System.IO.DirectoryInfo get_Parent()
get_Root                   Method         System.IO.DirectoryInfo get_Root()
InitializeLifetimeService Method          System.Object InitializeLifetimeService()
MoveTo                     Method         System.Void MoveTo(String destDirName)
Refresh                    Method         System.Void Refresh()
```

```
SetAccessControl          Method         System.Void
SetAccessControl(DirectorySecurity directorySecurity)
set_Attributes            Method         System.Void set_Attributes(FileAttributes
value)
set_CreationTime          Method         System.Void set_CreationTime(DateTime
value)
set_CreationTimeUtc       Method         System.Void set_CreationTimeUtc(DateTime
value)
set_LastAccessTime        Method         System.Void set_LastAccessTime(DateTime
value)
set_LastAccessTimeUtc     Method         System.Void set_LastAccessTimeUtc(DateTime
value)
set_LastWriteTime         Method         System.Void set_LastWriteTime(DateTime
value)
set_LastWriteTimeUtc      Method         System.Void set_LastWriteTimeUtc(DateTime
value)
ToString                  Method         System.String ToString()
PSChildName               NoteProperty   System.String PSChildName=mytest
PSDrive                   NoteProperty   System.Management.Automation.PSDriveInfo
PSDrive=C
PSIsContainer             NoteProperty   System.Boolean PSIsContainer=True
PSParentPath              NoteProperty   System.String
PSParentPath=Microsoft.PowerShell.Core\FileSystem::C:\
PSPath                    NoteProperty   System.String
PSPath=Microsoft.PowerShell.Core\FileSystem::C:\mytest
PSProvider                NoteProperty   System.Management.Automation.ProviderInfo
PSProvider=Microsoft.PowerShell.C...
Attributes                Property       System.IO.FileAttributes Attributes
{get;set;}
CreationTime              Property       System.DateTime CreationTime {get;set;}
CreationTimeUtc           Property       System.DateTime CreationTimeUtc {get;set;}
Exists                    Property       System.Boolean Exists {get;}
Extension                 Property       System.String Extension {get;}
FullName                  Property       System.String FullName {get;}
LastAccessTime            Property       System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc         Property       System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime             Property       System.DateTime LastWriteTime {get;set;}
LastWriteTimeUtc          Property       System.DateTime LastWriteTimeUtc {get;set;}
Name                      Property       System.String Name {get;}
Parent                    Property       System.IO.DirectoryInfo Parent {get;}
Root                      Property       System.IO.DirectoryInfo Root {get;}
Mode                      ScriptProperty System.Object Mode {get=$catr = "";...
```

From the listing of folder members, you can see there is a parent property. You can use the parent property information to find the genus of the mytest folder. This is shown here:

```
PS C:\> $a.parent

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d--hs         5/11/2007   2:39 PM            C:\
```

Perhaps you are interested in knowing when the folder was last accessed. To check on this, you can use the *LastAccessTime* property as shown here:

```
PS C:\> $a.LastAccessTime

Friday, May 11, 2007 2:39:12 PM
```

If you want to confirm the object contained in *$a* is indeed a folder, you can use the *PsIsContainer* property. The Get-Member output tells you that *PsIsContainer* is a Boolean value, and so it will reply as either true or false. This command is shown here:

```
PS C:\> $a.PsIsContainer
True
```

Maybe you would like to use one of the methods returned. You can use the *moveTo* method to move the folder to another location. Get-Member tells you that the *moveTo* method must have a string input that points to a destination directory. So, move the mytest folder to c:\movedFolder, then use the Test-Path cmdlet to check if the folder was moved to the new location. These commands are illustrated here:

```
PS C:\> $a.MoveTo("C:\movedFolder")
PS C:\> Test-Path c:\movedFolder
True
PS C:\> Test-Path c:\mytest
False
PS C:\>
```

To confirm the name of the folder you now have represented by the object in the *$a* variable, you can use the *Name* property. This is shown here with the associated output:

```
PS C:\> $a.name
movedFolder
```

If you want to delete the folder, you can use the *delete* method. This is shown here. To confirm it is actually deleted, use *dir m** to verify it is gone. These commands are shown here. Note that the folder has now been deleted.

```
PS C:\> $a.Delete()
PS C:\> dir m*


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         4/21/2007   4:56 PM            Maps
d----          5/5/2007   3:51 PM            music
-a---          2/1/2007   6:17 PM         54 MASK.txt
```

All of these commands and their associated output are contained in the Get-Member.txt file in the chapter01 folder on the companion CD-ROM.

Working with the .NET Framework

It might be interesting to note that these commands are actually commands that come from the .NET Framework. These are not Windows PowerShell commands at all. Of course the Get-Item, Get-Member, and Test-Path cmdlets are Windows PowerShell commands but System.IO.DirectoryInfo does not come from Windows PowerShell. This means you use the same methods and properties from Windows PowerShell as a professional developer using Visual Basic .NET or C#. This also means that much more information is available to you by using the Microsoft Developer Network (MSDN) and the Windows Software Development Kit (SDK). The good news for you: If you can't find information using the online help (by using Get-Help), you can always refer to the MSDN Web site or the Windows SDK for assistance.

Summary

This chapter examined the different ways to determine if Windows PowerShell is installed on a computer and the steps involved in configuring Windows PowerShell for use in a corporate enterprise environment. We covered the creation of Windows PowerShell profiles and explored various methods of launching both Windows PowerShell and Windows PowerShell commands. The chapter included extending the features of Windows PowerShell via the creation of custom aliases and functions. Finally, we concluded with a discussion of three Windows PowerShell cmdlets: Get-Help, Get-Command, and Get-Member.

Chapter 2

Scripting Windows PowerShell

After completing this chapter, you will be able to:

- Configure the scripting policy for Windows PowerShell.
- Run Windows PowerShell scripts.
- Use Windows PowerShell flow control statements.
- Use decision-making and branching statements.
- Identify and work with data types.
- Use regular expressions to provide advanced matching capabilities.
- Use command-line arguments.

> **On the Companion Disc** All the scripts used in this chapter are located on the CD that accompanies this book in the \scripts\chapter02 folder.

Why Use Scripting?

For many network administrators writing scripts—any kind of scripts—is a dark art more akin to reading tea leaves than administering a server. Indeed, while most large corporations seem to always have a "scripting guy," they rarely have more than one. This is in spite of the efforts by Microsoft to promote Visual Basic Scripting Edition (VBScript) as an administrative scripting language. While most professionals will agree that the ability to quickly craft a script to make ad hoc changes to dozens of networked servers is a valuable skill, few actually possess this skill. In reality, however, many of the corporate "scripting guy" skills are more akin to knowing where to find a script that can easily be modified than to actually understanding how to write a script from scratch.

Hopefully, this will change in the Windows PowerShell world. The Windows PowerShell syntax was deliberately chosen to facilitate ease of use and ease of learning. Corporate enterprise Windows administrators are the target audience.

So why use scripting? There are several reasons. First, a script makes it easy to document a particular sequence of commands. If you need to produce a listing of all the shares on a computer, you can use the *Win32_share* WMI class and the Get-WmiObject cmdlet to retrieve the results, as shown here:

```
PS C:\> Get-wmiObject win32_share
```

```
Name                    Path                         Description
----                    ----                         -----------
ADMIN$                  C:\Windows                   Remote Admin
C$                      C:\                          Default share
CCMLogs$                C:\Windows\system32\ccm\logs
CCMSetup$               C:\Windows\system32\ccmsetup
IPC$
Remote IPC
music                   C:\music                         none
VPCache$                C:\Windows\system32\VPCache
WMILogs$                C:\Windows\system32\wbem\logs
```

But, suppose you only want to have a list of file shares? You may not be aware that a file share is a type 0 share. So perhaps you need to search for this information on the Internet. Once you have obtained the information, use the modified command shown here:

```
PS C:\> Get-WmiObject win32_share -Filter "type = '0'"

Name                     Path                              Description
----                     ---                               -----------
CCMLogs$                 C:\Windows\system32\ccm\logs
CCMSetup$                C:\Windows\system32\ccmsetup
music                    C:\music                           none
VPCache$                 C:\Windows\system32\VPCache
WMILogs$                 C:\Windows\system32\wbem\logs
```

You can see that not only do you need to remember the share type of 0, but the syntax is a bit more complicated as well. So where do you write down this information? Here's one suggestion: When I was an administrator working on the Digital VAX, I kept a small pocket-size notebook to store such cryptic commands. Of course, if I ever lost my little notebook or failed to carry it, I was in big trouble!

Now suppose you are only interested in file shares that do not have a description assigned to them. This command is shown here:

```
PS C:\> Get-WmiObject win32_share -Filter "type = '0' AND description = ''"

Name                        Path                            Description
----                        ----                            -----------
CCMLogs$                    C:\Windows\system32\ccm\logs
CCMSetup$                   C:\Windows\system32\ccmsetup
VPCache$                    C:\Windows\system32\VPCache
WMILogs$                    C:\Windows\system32\wbem\logs
```

At this point, you may feel the command and associated syntax are complicated enough to justify writing a script. Creating the script is easy; simply copy it from the Windows PowerShell console and paste it into a text file. Name the script and change the extension to .ps1. You can then run the script from inside Windows PowerShell. The commands just shown are saved in Share.txt in the chapter02 folder on the companion CD-ROM. The script is named GetFileShares.ps1.

An additional advantage to configuring a command as a script is that you can easily make modifications. Whereas the previous command was limited to reporting only on file shares, you can make a change to the script to allow reporting on print shares, remote administrative shares, IPC shares, or any other defined share type. You can modify the script so you can choose a share type when you launch the script. To do this, use an *if ... else* statement to see if a command-line argument has been supplied to the script.

> **Tip** To check for a command-line argument, look for *$args*, which is the automatic variable created to hold command-line arguments.

If there is a command-line argument, use the value supplied to the command line. If no value is supplied when the script is launched, then you must supply a default value to the script. For this script, you will list file shares and inform the user that you are using default values. The Get-WmiObject syntax is the same as you used previously in the VBScript days. When writing a script, it's also useful to display a *usage string*. The following script, GetSharesWithArgs.ps1, includes an example command to assist you with typing the correct syntax for the script.

GetSharesWithArgs.ps1

```
if($args)
 {
   $type = $args
   Get-WmiObject win32_share -Filter "type = $type"
 }
ELSE
 {
  Write-Host
  "
  Using defaults values, file shares type = 0.
  Other valid types are:
  2147483651 for disk drive admin share
  2147483649 for print queue admin share
  2147483650 for device admin share
  2147483651 for ipc$ admin share
  Example: C:\GetSharesWithArgs.ps1 '2147483651'
  "
  $type = '0'
  Get-WmiObject win32_share -Filter "type = $type"
 }
```

Another reason why network administrators write Windows PowerShell scripts is to run the script as a scheduled task. In the Windows world there are multiple task scheduler engines. Using the *Win32_ScheduledJob* WMI class you can create, monitor, and delete scheduled jobs. This WMI class has been available since the Windows NT 4.0 days. Both Windows XP and Windows Server 2003 have the Schtasks.exe utility, which offers more flexibility than the *Win32_ScheduledJob* WMI class. Besides Schtasks.exe, Windows Vista and Windows Server 2008 also include the *Schedule.Service* object to simplify the configuration of scheduled jobs.

The script, ListProcessesSortResults.ps1, is something you may want to schedule to run several times daily. The script produces a list of currently running processes and writes the results to a text file as a formatted and sorted table.

ListProcessesSortResults.ps1

```
$args = "localhost","loopback","127.0.0.1"

foreach ($i in $args)
   {$strFile = "c:\mytest\"+ $i +"Processes.txt"
    Write-Host "Testing" $i "please wait ...";
    Get-WmiObject -computername $i -class win32_process |
    Select-Object name, processID, Priority, ThreadCount, PageFaults,
      PageFileUsage |
    Where-Object {!$_.processID -eq 0} | Sort-Object -property name |
    Format-Table | Out-File $strFile}
```

Configuring the Scripting Policy

Since scripting in Windows PowerShell is not enabled by default, it is important to verify the level of scripting support provided on the platform before deployment of either scripts or commands. If you attempt to run a Windows PowerShell script when the support has not been enabled, you'll receive an error message and the script won't run. This error message is shown in Figure 2-1.

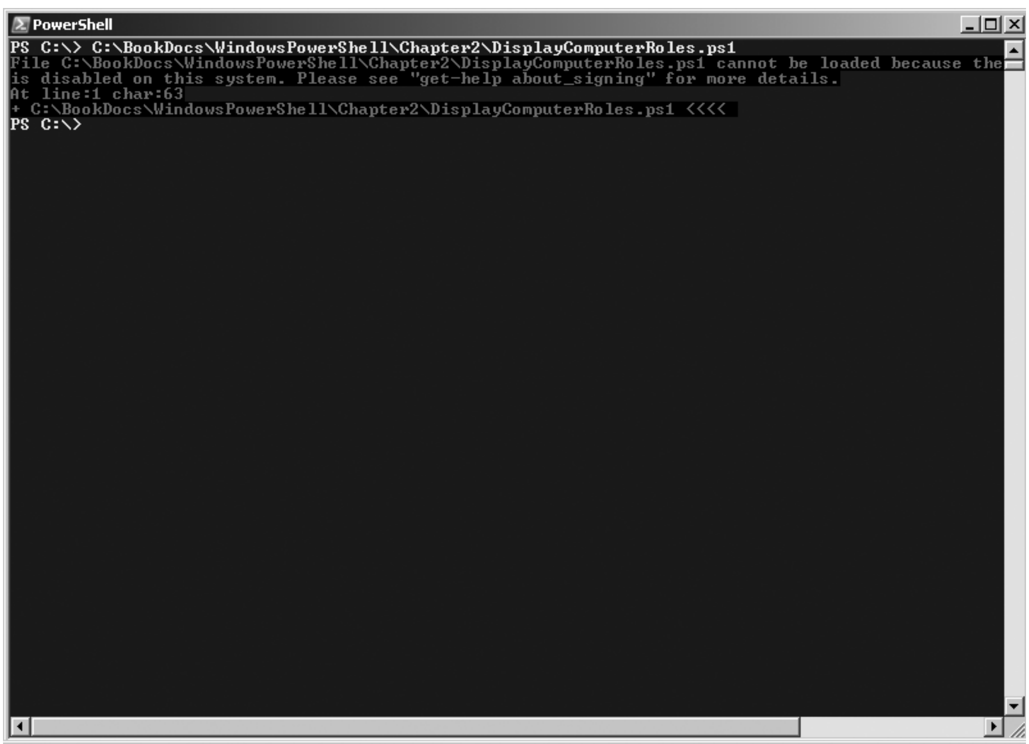

Figure 2-1 Attempting to run a script before scripting support is enabled generates an error.

This is referred to as the restricted execution policy. There are four levels of execution policy that can be configured in Windows PowerShell with the Set-ExecutionPolicy cmdlet. These four levels are listed in Table 2-1. The restricted execution policy can be configured via Group Policy by using the Turn On Script Execution Group Policy setting in Active Directory directory service. It can be applied to either the computer object or to the user object. The computer object setting takes precedence over other settings.

> **Tip** To retrieve the script execution policy use the Get-ExecutionPolicy cmdlet.

Configure user preferences for the restricted execution policy with the Set-ExecutionPolicy cmdlet but note that these preferences won't override settings configured by Group Policy. Obtain the resulting set of restricted execution policy settings by using the Get-ExecutionPolicy cmdlet.

Table 2-1 Script Execution Policy Levels

Level	Meaning
Restricted	Will not run scripts or configuration files.
AllSigned	All scripts and configuration files must be signed by a trusted publisher.
RemoteSigned	All scripts and configuration files downloaded from the Internet must be signed by a trusted publisher.
Unrestricted	All scripts and configuration files will run. Scripts downloaded from the Internet will prompt for permission prior to running.

You should be aware that on Windows Vista, access to the registry key that contains the script execution policy is restricted. A "normal" user will not be allowed to modify the key, and even an administrator running with User Account Control (UAC) turned on will not be allowed to modify the setting. If modification is attempted, the error shown in Figure 2-2 will be generated.

There are, of course, several ways around the UAC issue. One choice is to simply turn off UAC; in most circumstances this is an undesirable solution. A better solution is to right-click the Windows PowerShell icon and select Run As Administrator as shown in Figure 2-3.

If you find right-clicking a bit too time-consuming (as I do!) you might prefer to create a second Windows PowerShell shortcut. You might name this second shortcut admin_ps and configure the shortcut properties to launch with administrative rights. For about 90 percent of all your administrative needs, the first shortcut should suffice. If, however, you need "more power," then choose the administrative one. The shortcut properties you can use for the admin_ps "administrative PowerShell" shortcut are shown in Figure 2-4.

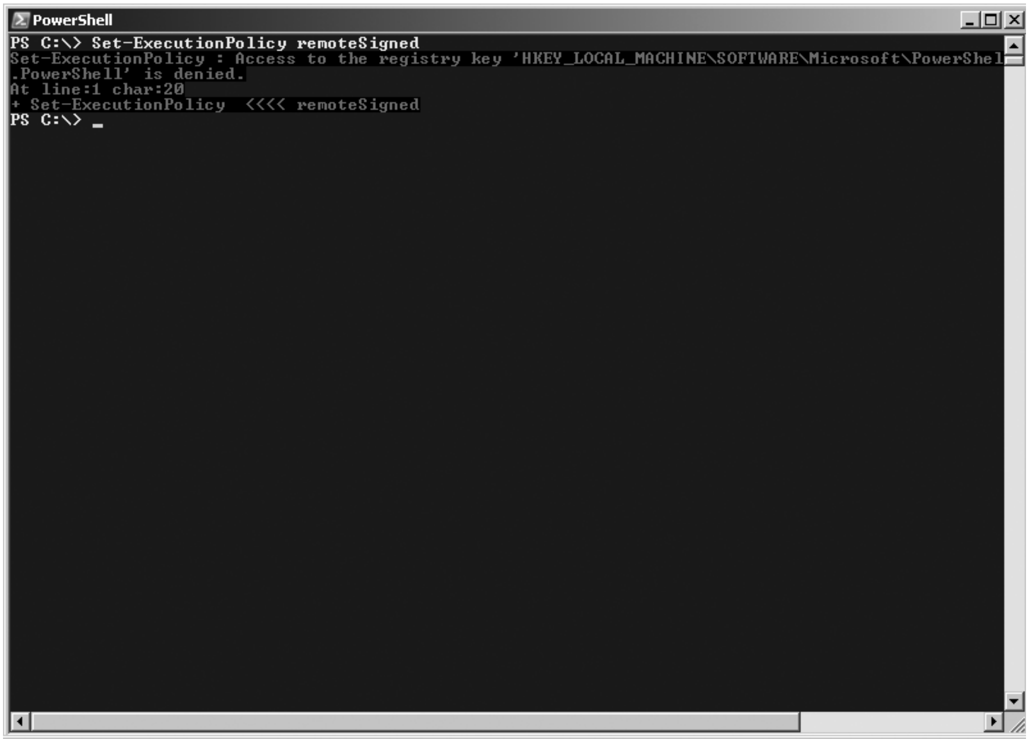

Figure 2-2 An attempt to run the Set-ExecutionPolicy cmdlet will fail if the user does not have administrative rights.

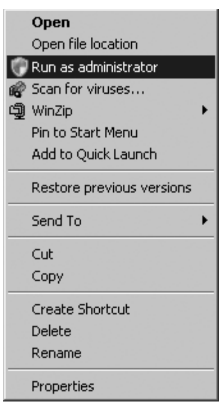

Figure 2-3 To launch Windows PowerShell with administrative rights, you can right-click the icon, and select Run As Administrator.

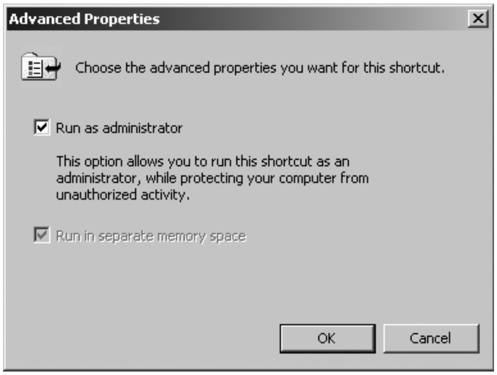

Figure 2-4 To configure the Windows PowerShell shortcut to run with administrative rights, choose the Run As Administrator check box found under Advanced Properties.

Running Windows PowerShell Scripts

You can't simply double-click a Windows PowerShell script and have it run. You cannot type the name in the Start | Run dialog box, either. If you are inside Windows PowerShell, you can run scripts if you have enabled the execution policy, but you need to type the entire path to the script you want to run and make sure to include the .ps1 extension.

If you need to run a script from outside Windows PowerShell, you must type the full path to the script, but you must also feed it as an argument to the PowerShell.exe program. In addition, you probably want to specify the *-noexit* switch so you can read the output from the script inside the Windows PowerShell console. This syntax is shown in Figure 2-5.

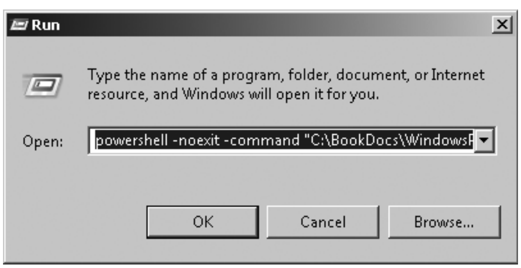

Figure 2-5 To run a Windows PowerShell script from outside the console, use the *-noexit* argument to allow you to see the results of the script.

Use of Variables

When working with Windows PowerShell, the default is that you don't need to declare variables prior to use; the variable is declared when you use it to hold data. All variable names must be preceded with a dollar sign. There are a number of special variables in Windows PowerShell. These variables are created automatically and each has a special meaning. Table 2-2 lists the special variables and their associated meanings.

Table 2-2 Use of Special Variables

Name	Use
*$^*	Contains the first token of the last line input into the shell.
$$	Contains the last token of the last line input into the shell.
$_	The current pipeline object; used in script blocks, filters, Where-Object, ForEach-Object, and *switch*.
$?	Contains the success/fail status of the last statement.
$args	Used in creating functions requiring parameters.
$error	If an error occurred, the *error* object is saved in the *$error* variable.
$executioncontext	The *execution* objects available to cmdlets.
$foreach	Refers to the enumerator in a *foreach* loop.
$home	The user's home directory; set to %HOMEDRIVE%\%HOMEPATH%.
$input	Input is piped to a function or code block.
$match	A hash table consisting of items found by the *-match* operator.
$myinvocation	Information about the currently executing script or command line.
$pshome	The directory where Windows PowerShell is installed.
$host	Information about the currently executing host.
$lastexitcode	The exit code of the last native application to run.
$true	Boolean TRUE.
$false	Boolean FALSE.
$null	A null object.
$this	In the Types.ps1 XML file and some script block instances this represents the current object.
$ofs	Output field separator used when converting an array to a string.
$shellid	The identifier for the shell. This value is used by the shell to determine the execution policy and what profiles are run at startup.
$stacktrace	Contains detailed stack trace information about the last error.

Use of Constants

Constants in Windows PowerShell are like variables with two important exceptions: Their value never changes, and they cannot be deleted. Constants are created by using the Set-Variable cmdlet and specifying the *-option* argument to be equal to constant.

> **Tip** When referring to a constant in the body of the script, you must preface it with the dollar sign—just like any other variable. However, when creating the constant (or even a variable) by using the Set-Variable cmdlet, as you specify the *name* argument you don't include a dollar sign.

In the GetHardDiskDetails.ps1 script that follows, there is a constant named *$intDriveType* with a value of 3 assigned. This constant is used because the *Win32_LogicalDisk* WMI class uses a value of 3 in the *DiskType* property to describe a local fixed disk. When using Where-Object and a value of 3, you eliminate network drives, removable drives, and ram drives from the items returned.

The *$intDriveType* constant is only used with the Where filter line. The value of *$strComputer*, however, will change once for each computer name that is specified in the array *$aryComputers*. In the GetHardDiskDetails.ps1 script, the value of *$strComputer* will change twice. The first time through the loop it will be equal to *loopback* and the second time through the loop it will be equal to *localhost*. Even if you add 250 different computer names, the effect will be the same—the value of *$strComputer* will change each time through the loop.

GetHardDiskDetails.ps1

```
$aryComputers = "loopback", "localhost"
Set-Variable -name intDriveType -value 3 -option constant

foreach ($strComputer in $aryComputers)

   {"Hard drives on: " + $strComputer
   Get-WmiObject -class win32_logicaldisk -computername $strComputer|
      Where {$_.drivetype -eq $intDriveType}}
```

Using Flow Control Statements

Once scripting support is enabled on Windows PowerShell, you have access to some advanced flow control cmdlets. However, this does not mean you cannot do flow control inside the console. You can certainly use flow control statements inside the console. This is shown here:

```
PS C:\> Get-Process | foreach ( $_.name ) { if ( $_.name -eq "system" ) {
Write-Host "system process is ID : " $_.ID } }
```

The problem is the amount of typing. It may be preferable to save such a command in a script. Besides saving a long command in a file, there is also an advantage in readability. For example, you can line up the curly brackets and the other components of the commands. You can also avoid hard-coding process names into the script and instead save them as variables. This makes it easy to modify the script or even to write the script to accept command-line arguments. In the GetProcessByID.ps1 script shown here, you can see these options exhibited.

GetProcessByID.ps1

```
$strProcess = "system"
Get-Process |
foreach ( $_.name ) {
  if ( $_.name -eq $strProcess )
   {
    Write-Host "system process is ID : " $_.ID
   }
}
```

Adding Parameters to ForEach-Object

In the GetWmiAndQuery.ps1 script, the ForEach-Object cmdlet produces a listing from all the WMI classes that have names containing *usb*. This particular script is very useful in that it produces a listing of both the process name and associated process ID (PID). In addition, the GetProcessByID.ps1 script is a good candidate to modify to accept a command-line argument. Begin with the *list* switch from the Get-WmiObject cmdlet; you'll end up with a complete listing of all WMI classes in the default WMI namespace. Pipeline the resulting object into the Where-Object cmdlet and filter the result set by the *Name* property when it is like the value contained in the variable *$strClass*.

Using the *Begin* Parameter

Use the *-begin* parameter of the ForEach-Object cmdlet to write the name used to generate the WMI class listings. This action does not affect the current pipeline object. In fact, neither the *-begin* parameter or the *-end* parameter interact with the current pipeline object. But they are great places to perform pre-processing and post-processing. The *-process* parameter is used to contain the script block that will interact with the current pipeline object. This is the default parameter, and doesn't need to be named. The Get-WmiAndQuery.ps1 script is shown here.

GetWmiAndQuery.ps1

```
$strClass = "usb"
Get-WmiObject -List |
Where { $_.name -like "*$strClass*" } |
ForEach-Object -begin `
  {
    Write-Host "$strClass wmi listings"
   Start-Sleep 3
   } `
-Process `
   {
     Get-wmiObject $_.name
   }
```

In the ProcessUsbHub.ps1 script, the Get-WmiObject cmdlet retrieves instances of the *Win32_USBHub* class. Once we have a collection of *usb hub* objects, we pipeline the object to the ForEach-Object cmdlet. Suggestion: To make the script easier to read, line up all the *-begin*, *-process,* and *-end* parameters on the left side of the script. However, you will have to use the "backtick" or grave accent (`) to indicate line continuation.

> **Tip** The environment variable *%computername%* is always available and can be used to extract the computer name for a script. An easy way to retrieve the value of this variable is to use the Get-Item cmdlet to grab the value from the env:\ psdrive. The *Value* property contains the computer name. This is illustrated here: (Get-Item env:\computerName) value.

The *-begin* section uses a code block to write the name of computer using the Write-Host cmdlet. Use a sub-expression to get the computer name from the env:\ psdrive; use the %computername% variable and extract its value.

Using the *Process* Parameter

In the *-process* section, simply use the current pipeline object (indicated by the $_ automatic variable) to print the *PnpDeviceID* property from the *Win32_USBHub* WMI class. Again, use the grave accent to indicate line continuation.

Using the *End* Parameter

The last section of the ProcessUsbHub.ps1 script contains the *-end* parameter. Use the Write-Host cmdlet to print a string that indicates the command completed, and use a sub-expression to print the value returned by the Get-Date cmdlet. The ProcessUsbHub.ps1 script is listed here.

ProcessUsbHub.ps1

```
Get-WmiObject win32_usbhub |
foreach-object `
-begin { Write-Host "Usb Hubs on:" $(Get-Item env:\computerName).value } `
-process { $_.pnpDeviceID}  `
-end { Write-Host "The command completed at $(get-date)" }
```

Using the *For* Statement

Similar to the ForEach-Object cmdlet, the *for* statement is used to control execution of a script block as long as a condition is true. Most of the time, you will use the *for* statement to perform an action a certain number of times. In the line of code that follows, notice the basic *for* construction. Use parentheses to separate the expression being evaluated from the code block contained in curly brackets. The evaluated expression is composed of three sections. The first section is a variable *$a*; you assign the value of 1 to it. The second section contains the condition to be evaluated. In the code shown here, as long as the variable *$a* is less than or equal to the number 3, the command in the code block section continues to run. The last section of the evaluation expression adds the number 1 to the variable *$a*. The code block is a simple printout of the word *hello*.

```
for ($a = 1; $a -le 3 ; $a++) {"hello"}
```

The PingARange.ps1 script shown here is a very useful little script because it can be used to ping a range of Internet protocol (IP) addresses and will tell you whether or not the computer is responding to Internet Control Message Protocol (ICMP) packets. This is helpful in doing network discovery or in ensuring a computer is talking to the network. The *$intPing* variable is set to 10 and defined as an integer. Next, the *$intNetwork* variable is assigned the string 127.0.0. and is defined as a string.

The *for* statement is used to execute the remaining code the number of times specified in the *$intPing* variable. The counter variable is created on the *for* statement line. This counter variable, named *$i*, is assigned the value of 1. As long as *$i* is less than or equal to the value set in the *$intPing* variable, the script will continue to execute. The final step, completed inside the evaluator section of the *for* statement, is to add one to the value of *$i*.

The code block begins with the curly bracket. Inside the code block, first create a variable named *$strQuery*; this is the string that holds the WMI query. Placing this in a separate variable makes it easier to use *$intNetwork* along with the *$i* counter variable; these are used to create a valid IP address for the WMI query that results in a ping.

The *$wmi* variable is used to hold the collection of objects that is returned by the Get-WmiObject cmdlet. By using the *optional query* argument of the Get-WmiObject cmdlet, you are able to supply a WMI query. The *StatusCode* property contains the result of the ping operation. A 0 indicates success, any other number means the ping failed. To present this information in a clear fashion, use an *if ... else* statement to evaluate the *StatusCode* property.

PingARange.ps1

```
 [int]$intPing = 10
[string]$intNetwork = "127.0.0."

for ($i=1;$i -le $intPing; $i++)
{
$strQuery = "select * from win32_pingstatus where address = '" +
$intNetwork + $i + "'"
   $wmi = get-wmiobject -query $strQuery
   "Pinging $intNetwork$i ... "
   if ($wmi.statuscode -eq 0)
      {"success"}
      else
         {"error: " + $wmi.statuscode + " occurred"}
```

Using Decision-Making Statements

The ability to make decisions to control branching in a script is a fundamental technique. In fact, this is the basis of automation. A condition is detected and evaluated, and a course of action is determined. If you are able to encapsulate your logic into a script, you are well on your way to having servers that monitor themselves. As an example, when you open Task Manager on the server, what is the first thing you do? I often sort the list of processes by memory consumption. The GetTopMemory.ps1 script, shown here, does this.

GetTopMemory.ps1

```
Get-Process |
Sort-Object workingset -Descending |
Select-Object -First 5
```

The GetTopMemory.ps1 script might be useful because it saves time in sorting a list. But what do you do next? Do you kill the top memory consuming process? If you do, then there is no decision to make. However, suppose you want to kill off only user mode processes that consume more than 100 MB of memory? That may be a more constructive and better choice. This will require some decision-making capability. Let us first examine the classic *if … elseif … else* decision structure.

Using *If … Elseif … Else*

The most basic decision-making statement is the *if … elseif … else* structure. This structure is easy to use because it is perfectly natural and is implied in normal conversation. For example, consider the following conversation between two American tourists in Copenhagen:

```
If ( sunny and warm )
  { go to NyHavn }
Elseif ( cloudy and cool )
  { go to Tivoli }
Else
  { take s-tog to Malmo }
```

Even if you don't speak Danish, you will be able to follow the conversation. If it is sunny and warm, then the tourists will go to NyHavn. The first condition evaluation is whether the weather is going to be sunny and warm. The condition is always enclosed in smooth parentheses. The script block that will be executed if the condition is true is in curly brackets. In this example, if the weather is sunny and warm, the tourists will go to NyHavn (a beautiful port with lots of outdoor cafes). However, if the weather is cloudy and cool, they will go to Tivoli (an amusement park in the center of Copenhagen). If neither of these conditions is true, for example, if it is raining or snowing, the tourists will take the train to Malmo (a city in Sweden famous for its shopping).

To use the GetServiceStatus.ps1 script, you will first obtain a listing of all the services on the computer. Do this by using the Get-Service cmdlet. Once you have a listing of the services, use the Sort-Object cmdlet to sort the list of services based on their status. Next, use *foreach* to walk through the collection of services. As you iterate through the services, use *if … elseif … else* to evaluate the status. If the service is stopped, use the color red to display the name and status. If the service is running, use green to display the name and status. If the service is in a different state (such as pause), default to yellow to display the name and status. A decision matrix such as this is very useful in allowing you to quickly scan a long list of services. The GetServiceStatus.ps1 script is shown here. The constant color values that can be used with the Write-Host cmdlet are detailed in the table that follows.

GetServiceStatus.ps1

```
Get-Service |
Sort-Object status -descending |
foreach {
  if ( $_.status -eq "stopped")
   {Write-Host $_.name $_.status -ForegroundColor red}
```

```
  elseif ( $_.status -eq "running" )
   {Write-Host $_.name $_.status -ForegroundColor green}
  else
   {Write-Host $_.name $_.status -ForegroundColor yellow}
}
```

Black	DarkBlue	DarkGreen	DarkCyan
DarkRed	DarkMagenta	DarkYellow	Gray
DarkGray	Blue	Green	Cyan
Red	Magenta	Yellow	White

Using *Switch*

In other programming languages, *switch* would be called the *select case* statement. The *switch* statement is used to evaluate a condition against a series of potential matches. In this way, it is essentially a streamlined *if … elseif* statement. When using the *switch* statement, the condition to be evaluated is contained in side parentheses. Then, each condition to be evaluated is placed inside a curly bracket within the code block. This is shown in the following command:

```
$a=5;switch ($a) { 4{"four detected"} 5{"five detected"} }
```

In the DisplayComputerRoles.ps1 script that follows, the script begins by using the *$wmi* variable to hold the object that is returned by using the Get-WmiObject cmdlet. The *DomainRole* property of the *Win32_computersystem* class is returned as a coded value. To produce an output that is more readable, the *switch* statement is used to match the value of the *DomainRole* property to the appropriate text value.

DisplayComputerRoles.ps1

```
$wmi = get-wmiobject win32_computersystem
"computer " + $wmi.name + " is: "
switch ($wmi.domainrole)
   {
   0 {"`t Stand alone workstation"}
   1 {"`t Member workstation"}
   2 {"`t Stand alone server"}
   3 {"`t Member server"}
   4 {"`t Back up domain controller"}
   5 {"`t Primary domain controller"}
   default {"`t The role can not be determined"}
   }
```

Evaluating Command-Line Arguments

Switch is ideally suited to evaluate command-line arguments. In the GetDriveArgs.ps1 script example that follows, you can use a function named *funArg* to evaluate the value of the automatic variable *$args*. This automatic variable contains arguments supplied to the command line when a script is run. This is a convenient variable to use when working with command-line arguments. *Switch* is used to evaluate the value of *$args*. Four parameter arguments are allowed with this script. The *all* argument does a WMI query to retrieve basic information on all logical disks on the computer. The argument *c* is used to return only information about the C drive. An interesting trick: The floppy drive is typically enumerated first, and the second element in the array is the C drive. If this is not the case on your system, you can change it. The purpose of the script is simply to point out the use of *switch* to parse command-line arguments. Using the array element number is a nice way to retrieve WMI information in Windows PowerShell. The *free* argument is used to only return free disk space on the C drive.

The *help* argument is used to print a help statement. It uses a here-string to make it easy to type in the help message. The help message displays the purpose of the script and several examples of command lines.

GetDriveArgs.ps1

```
Function funArg()
{
switch ($args)
{
 "all" { gwmi win32_logicalDisk }
 "c"   { (gwmi win32_logicaldisk)[1] }
 "free" { (gwmi win32_logicaldisk)[1].freespace }
 "help" { $help = @"
This script will print out the drive information for
All drives, only the c drive, or the free space on c:
It also will print out a help topic
EXAMPLE:
 >GetDriveArgs.ps1 all
   Prints out information on all drives
 >GetDriveArgs.ps1 c
   Prints out information on only the c drive
 >GetDriveArgs.ps1 free
   Prints out freespace on the c drive
"@ ; Write-Host $help }
}
}

#$args = "help"
funArg($args)
```

Using *Switch* Wildcards

One of the more interesting uses of the *switch* command is the use of wildcards. This can open up new opportunities to write clear and compact code that is both powerful and easy to implement. The SwitchIPConfig.ps1 script holds the results of the *ipconfig /all* command in the *$a* variable. Use *switch* with the *-wildcard* argument and feed it the text to parse inside the smooth parenthesis. Then, open the script block with the curly brackets and type the pattern to match. In this case, it is a simple *DHCP Server* phrase. In the script block that will execute when the pattern match is found, use the Write-Host cmdlet to print the current line inside the *switch* block. The interesting point is the use of the *$switch* automatic variable as the enumerator. Specify the current property and retrieve the current line that is processing. In this way, you can print the line you are interested in examining. The SwitchIPConfig.ps1 script is shown here.

SwitchIPConfig.ps1

```
$a = ipconfig /all

switch -wildCard ($a)
 {
   "*DHCP Server*" { Write-Host $switch.current }
 }
```

Using *Switch* with Regular Expressions

Unlike a normal *select case* statement, the *switch* statement has the ability to work with regular expressions. When looking for valuable information, you can use the *switch* statement to open a text file, read the file into memory, and then use regular expressions to parse the file. Regular expressions can be as simple as matching a particular word or phrase or as complicated as validating a legitimate e-mail address. The SwitchRegEx.ps1 script that follows examines a sample text file for two words: *test* and *good*. If either word is found, the entire line containing the matched word prints.

Following the *switch* statement, you can use the *-regex* parameter to indicate that you want to use regular expressions as the matching tool. The value to switch on, inside the smooth parentheses, is actually a sub-expression that opens and reads the text file. The $ in front of the curly brackets surrounding the path to a text file is the command to open and read the text file into memory. Open the switch with the curly brackets and place each pattern to match inside single quotations. The code block that will execute if the regular expression is matched is also contained in curly brackets, and in this example it is a simple write-host. Once again, use the *$switch* enumerator to retrieve the current line where the pattern match occurs.

SwitchRegEx.ps1

```
switch -regex (${c:\testa.txt})
{
 'test' {Write-Host $switch.current}
 'good' {Write-Host $switch.current}
}
```

The text of the TestA.txt file is shown here. This example will assist you in evaluating the output from the script.

TestA.txt

```
This was a test file.
This was a good file.
This was a good test file.
```

Perhaps a more useful example of using the regular expression feature of the *switch* statement is the VersionOfVista.ps1 script. Assign the string *version* to the *$strPattern* variable, and hold the output of the net config workstation command in the *$text* variable. Then, use the *-regex* parameter on the *switch* statement and feed it the content stored in the *$text* variable, and look for the pattern that is stored in the *$strPattern* variable. Once you find it, print the entire line by using the current property of the automatic variable *$switch*. The nice thing about this script is that it tells you what version of Windows Vista you have. The entire output from net config workstation command is 19 lines long. To compare results, here is a sample output from VersionOfVista.ps1:

```
Software version                    Windows Vista (TM) Enterprise
```

VersionOfVista.ps1

```
$strPattern = "version"
$text = net config workstation

switch -regex ($text)
{
  $strPattern { Write-Host $switch.current }
}
```

Working with Data Types

Windows PowerShell is a strongly typed language that acts as if it were typeless. This is because Windows PowerShell does a good job of detecting data types and acting on them accordingly. If something appears to be a string, Windows PowerShell will treat it as a string. As an example, consider these three statements:

```
PS C:\> 1 + 1
2
PS C:\> 12:00 + :30
Unexpected token ':00' in expression or statement.
At line:1 char:6
+ 12:00  <<<< + :30
PS C:\> a + b
The term 'a' is not recognized as a cmdlet, function, operable program,
or script file. Verify the term and try again At line:1 char:2 + a  <<<< + b
PS C:\>
```

Notice that only one statement completed without error—the one containing 1 + 1. Windows PowerShell properly detected these as numbers and allowed the addition to proceed. However, it is impossible to add letters or time.

However, if you put the letters *a* and *b* within double quotation marks and then add them, you will notice that the action succeeds. This is shown here:

```
PS C:\> "a" + "b"
Ab
```

This behavior is not surprising, and in fact, is to be expected. Double quotation marks turn the letters *a* and *b* into string values and concatenates the two letters. You can see this if you pipeline the letter *a* into the Get-Member cmdlet as shown here. Notice that the first line of output indicates the letter *a* is an object of the type *system.string*. Also observe that there are many properties and methods you can use on a *system.string* object.

```
PS C:\> "a" | get-member


   TypeName: System.String

Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone()
System.Int32 CompareTo(String strB)
Contains         Method               System.Boolean Contains(String value)
CopyTo           Method               System.Void CopyTo(Int32 sourceIndex, Char[]
destination, Int32 destinationIn
EndsWith         Method               System.Boolean EndsWith(String value),
System.Boolean EndsWith(String value,
Equals           Method               System.Boolean Equals(Object obj),
System.Boolean Equals(String value), Syste...
GetEnumerator    Method               System.CharEnumerator GetEnumerator()
GetHashCode      Method               System.Int32 GetHashCode()
GetType          Method               System.Type GetType()
GetTypeCode      Method               System.TypeCode GetTypeCode()
get_Chars        Method               System.Char get_Chars(Int32 index)
get_Length       Method               System.Int32 get_Length()
IndexOf          Method               System.Int32 IndexOf(Char value, Int32
startIndex, Int32 count), System.Int32...
IndexOfAny       Method               System.Int32 IndexOfAny(Char[] anyOf, Int32
startIndex, Int32 count), System....
Insert           Method               System.String Insert(Int32 startIndex, String
value)
IsNormalized     Method               System.Boolean IsNormalized(), System.Boolean
IsNormalized(NormalizationForm
LastIndexOf      Method               System.Int32 LastIndexOf(Char value, Int32
startIndex, Int32 count), System.I...
LastIndexOfAny   Method               System.Int32 LastIndexOfAny(Char[] anyOf, Int32 start
Index, Int32 count), Sys...
Normalize        Method               System.String Normalize(), System.String
Normalize(NormalizationForm normaliz...
PadLeft          Method               System.String PadLeft(Int32 totalWidth),
System.String PadLeft(Int32 totalWid...
PadRight         Method               System.String PadRight(Int32 totalWidth),
System.String PadRight(Int32 totalW...
Remove           Method               System.String Remove(Int32 startIndex, Int32
count), System.String Remove(Int...
Replace          Method               System.String Replace(Char oldChar, Char
newChar), System.String Replace(Stri...
Split            Method               System.String[] Split(Params Char[]
separator), System.String[] Split(Char[] ...
StartsWith       Method               System.Boolean StartsWith(String value),
System.Boolean StartsWith(String val...
Substring        Method               System.String Substring(Int32 startIndex),
System.String Substring(Int32 star...
```

```
ToCharArray      Method                System.Char[] ToCharArray(), System.Char[]
ToCharArray(Int32 startIndex, Int3...
ToLower          Method                System.String ToLower(), System.String
ToLower(CultureInfo culture)
ToLowerInvariant Method                System.String ToLowerInvariant()
ToString         Method                System.String ToString(), System.String
ToString(IFormatProvider provider)
ToUpper          Method                System.String ToUpper(), System.String
ToUpper(CultureInfo culture)
ToUpperInvariant Method                System.String ToUpperInvariant()
Trim             Method                System.String Trim(Params Char[] trimChars),
System.String Trim()
TrimEnd          Method                System.String TrimEnd(Params Char[]
trimChars)
TrimStart        Method                System.String TrimStart(Params Char[]
trimChars)
Chars            ParameterizedProperty System.Char Chars(Int32 index) {get
```

If you pipeline the number 1 into the Get-Member cmdlet, you will see that it is a *system.int32* object, with a smaller listing of methods available than is available with the string class:

```
PS C:\> 1 | get-member


   TypeName: System.Int32

Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     System.Int32 CompareTo(Int32 value), System.Int32
CompareTo(Object value)
Equals      Method     System.Boolean Equals(Object obj), System.Boolean
Equals(Int32 obj)
GetHashCode Method     System.Int32 GetHashCode()
GetType     Method     System.Type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
ToString    Method     System.String ToString(), System.String
ToString(IFormatProvider provider), System.String ToS...
```

Once you have figured out how to use Get-Member to verify the reason for the behavior of an object, you can use the *type constraint* objects to confirm an object of a specific data type. If you want 12:00 to be interpreted as a *date time* object, use the [datetime] type constraint to cast the string 12:00 into a *date time* object. This is shown here:

```
PS C:\> [datetime]"12:00" | get-member


   TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(TimeSpan value)
AddDays              Method         System.DateTime AddDays(Double value)
AddHours             Method         System.DateTime AddHours(Double value)
AddMilliseconds      Method         System.DateTime AddMilliseconds(Double value)
AddMinutes           Method         System.DateTime AddMinutes(Double value)
AddMonths            Method         System.DateTime AddMonths(Int32 months)
```

```
AddSeconds           Method         System.DateTime AddSeconds(Double value)
AddTicks             Method         System.DateTime AddTicks(Int64 value)
AddYears             Method         System.DateTime AddYears(Int32 value)
CompareTo            Method         System.Int32 CompareTo(Object value),
System.Int32 CompareTo(DateTime value)
Equals               Method         System.Boolean Equals(Object value),
System.Boolean Equals(DateTime value)
GetDateTimeFormats   Method         System.String[] GetDateTimeFormats(),
System.String[] GetDateTimeFormats(IFormat...
GetHashCode          Method         System.Int32 GetHashCode()
GetType              Method         System.Type GetType()
GetTypeCode          Method         System.TypeCode GetTypeCode()
get_Date             Method         System.DateTime get_Date()
get_Day              Method         System.Int32 get_Day()
get_DayOfWeek        Method         System.DayOfWeek get_DayOfWeek()
get_DayOfYear        Method         System.Int32 get_DayOfYear()
get_Hour             Method         System.Int32 get_Hour()
get_Kind             Method         System.DateTimeKind get_Kind()
get_Millisecond      Method         System.Int32 get_Millisecond()
get_Minute           Method         System.Int32 get_Minute()
get_Month            Method         System.Int32 get_Month()
get_Second           Method         System.Int32 get_Second()
get_Ticks            Method         System.Int64 get_Ticks()
get_TimeOfDay        Method         System.TimeSpan get_TimeOfDay()
get_Year             Method         System.Int32 get_Year()
IsDaylightSavingTime Method         System.Boolean IsDaylightSavingTime()
Subtract             Method         System.TimeSpan Subtract(DateTime value),
System.DateTime Subtract(TimeSpan value)
ToBinary             Method         System.Int64 ToBinary()
ToFileTime           Method         System.Int64 ToFileTime()
ToFileTimeUtc        Method         System.Int64 ToFileTimeUtc()
ToLocalTime          Method         System.DateTime ToLocalTime()
ToLongDateString     Method         System.String ToLongDateString()
ToLongTimeString     Method         System.String ToLongTimeString()
ToOADate             Method         System.Double ToOADate()
ToShortDateString    Method         System.String ToShortDateString()
ToShortTimeString    Method         System.String ToShortTimeString()
ToString             Method         System.String ToString(), System.String
ToString(String format), System.String T...
ToUniversalTime      Method         System.DateTime ToUniversalTime()
Date                 Property       System.DateTime Date {get;}
Day                  Property       System.Int32 Day {get;}
DayOfWeek            Property       System.DayOfWeek DayOfWeek {get;}
DayOfYear            Property       System.Int32 DayOfYear {get;}
Hour                 Property       System.Int32 Hour {get;}
Kind                 Property       System.DateTimeKind Kind {get;}
Millisecond          Property       System.Int32 Millisecond {get;}Property
 System.Int32 Minute {get;}
Month                Property       System.Int32 Month {get;}
Second               Property       System.Int32 Second {get;}
Ticks                Property       System.Int64 Ticks {get;}
TimeOfDay            Property       System.TimeSpan TimeOfDay {get;}
Year                 Property       System.Int32 Year {get;}
DateTime             ScriptProperty System.Object DateTime {get=if
($this.DisplayHint -ieq  "Date")...
```

There is no reason to use Get-Member to determine the data type of a particular object if you are only interested in the name of the object. To do this, you can use the *getType()* method as shown here. In the first case, you confirm that 12:00 is indeed a string. In the second case, you cast the string into a *datetime* data type, and confirm it by once again using the *getType()* method as shown here:

```
PS C:\> "12:00".getType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object


PS C:\> ([dateTime]"12:00").getType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DateTime                                 System.ValueType
```

All of these commands are in the DataTypes.txt file found in the chapter02 folder on the companion CD-ROM. Additional data type aliases are shown in Table 2-3.

Table 2-3 Data Type Aliases

Alias	Type
[int]	32-bit signed integer
[long]	64-bit signed integer
[string]	Fixed length string of Unicode characters
[char]	A Unicode 16-bit character
[bool]	True/False value
[byte]	An 8-bit unsigned integer
[double]	Double-precision 64-bit floating point number
[datetime]	DateTime data type
[decimal]	A 128-bit decimal value
[single]	Single precision 32-bit floating point number
[array]	An array of values
[xml]	*Xml* objects
[hashtable]	A *hashtable* object (similar to a *dictionary* object)

Unleashing the Power of Regular Expressions

One of the interesting features of Windows PowerShell is the ability to work with regular expressions. Regular expressions are optimized to manipulate text. You've learned about using regular expressions with the *switch* statement to match a particular word, however, you can do as much with the *-wildcard* switch. Now you'll learn some of the more advanced tasks you can complete with regular expressions. Table 2-4 lists the escape sequences you can use with regular expressions.

Table 2-4 Escape Sequences

Character	Description
ordinary characters	Characters other than . $ ^ { [(|) * + ? \ match themselves.
\a	Matches a bell (alarm) \u0007.
\b	Matches a backspace \u0008 if in a [] character class; in a regular expression, *\b* is a word boundary.
\t	Matches a tab \u0009.
\r	Matches a carriage return \u000D.
\v	Matches a vertical tab \u000B.
\f	Matches a form feed \u000C.
\n	Matches a new line \u000A.
\e	Matches an escape \u001B.
\040	Matches an ASCII character as octal (up to three digits); numbers with no leading zero are backreferences if they have only one digit or if they correspond to a capturing group number. For example, the character \040 represents a space.
\x20	Matches an ASCII character using hexadecimal representation (exactly two digits).
\cC	Matches an ASCII control character; for example, \cC is control-C.
\u0020	Matches a Unicode character using hexadecimal representation (exactly four digits).

The RegExTab.ps1 script illustrates using an escape sequence in a regular expression script. It opens a text file and looks for tabs. The easiest way to work with regular expressions is to store the pattern in its own variable. This makes it easy to modify and to experiment without worrying about breaking the script (simply use the # sign to comment out the line, then create a new line with the same name and a different value).

The RegExTab.ps1 script specifies \t as the pattern. According to Table 2-4 this means you look for tabs. Feed the pattern, contained in *$strPattern*, to the [regex] type accelerator as shown here:

```
$regex = [regex]$strPattern
```

Next, store the content of the TabLine.txt text tile into the *$text* variable by using the syntax shown here:

```
$text = ${C:\Chapter02\tabline.txt}
```

Then, use the *matches* method to parse the text file and look for matches with the pattern specified in the *$strPattern*. Notice that you have already associated the pattern with the *regular expression* object in the *$regex* variable. Count the number of times you have a match. The complete RegExTab.ps1 script is shown here.

RegExTab.ps1

```
$strPattern = "\t"
$regex = [regex]$strPattern

$text = ${C:\Chapter02\tabline.txt}

$mc = $regex.matches($text)
$mc.count
```

Table 2-5 lists the character patterns that can be used with regular expressions for performing advanced pattern matching.

Table 2-5 Character Patterns

Character	Description
[character_group]	Matches any character in the specified character group. For example, to specify all vowels, use [aeiou]. To specify all punctuation and decimal digit characters, use [\p{P}\d].
[^character_group]	Matches any character not in the specified character group. For example, to specify all consonants, use [^aeiou]. To specify all characters except punctuation and decimal digit characters, use [^\p{P}\d].
[firstCharacter-lastCharacter]	Matches any character in a range of characters. For example, to specify the range of decimal digits from '0' through '9', the range of lowercase letters from 'a' through 'f', and the range of uppercase letters from 'A' through 'F', use [0-9a-fA-F].
.	Matches any character except \n. If modified by the Singleline option, a period matches any character.
\p{name}	Matches any character in the Unicode general category or named block specified by name (for example, Ll, Nd, Z, IsGreek, and IsBoxDrawing).
\P{name}	Matches any character not in Unicode general category or specified named block
\w	Matches any word character. Equivalent to the Unicode general categories [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]. If ECMA-Script-compliant behavior is specified with the ECMAScript option, \w is equivalent to [a-zA-Z_0-9].
\W	Matches any nonword character. Equivalent to the Unicode general categories [^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]. If ECMAScript-compliant behavior is specified with the ECMAScript option, \W is equivalent to [^a-zA-Z_0-9].
\s	Matches any white-space character. Equivalent to the escape sequences and Unicode general categories [\f\n\r\t\v\x85\p{Z}]. If ECMAScript-compliant behavior is specified with the ECMA-Script option, \s is equivalent to [\f\n\r\t\v].

Table 2-5 Character Patterns *(continued)*

Character	Description
\S	Matches any non-white-space character. Equivalent to the escape sequences and Unicode general categories [^\f\n\r\t\v\x85\p{Z}]. If ECMAScript-compliant behavior is specified with the ECMAScript option, \S is equivalent to [^ \f\n\r\t\v].
\d	Matches any decimal digit. Equivalent to \p{Nd} for Unicode and [0-9] for non-Unicode, ECMAScript behavior.
\D	Matches any nondigit character. Equivalent to \P{Nd} for Unicode and [^0-9] for non-Unicode, ECMAScript behavior.

Suppose you want to identify white space in a file. To do this, you can use the match pattern \s which is listed in Table 2-5 as a character pattern. The ability to find white space in a text file is quite useful, because for many items, the end of line separator is just white space. To illustrate working with white space, examine the following RegWhiteSpace.ps1 script.

The first line of the script includes a line of text to use for testing against. The pattern comes from Table 2-5 and is a simple \s, which tells the regular expression you want to match on white space. Then use the *$matches* variable to hold the *match* object returned by the *match static* method of the regex type accelerator.

After printing the results of the match, move to phase two, which is to replace, using the same pattern. To do this, feed the pattern to the *replace* method along with the variable containing the unadulterated text message. Go ahead and print the value of *$strReplace* that now contains the modified object.

RegWhiteSpace.ps1

```
$strText = "a nice line of text. We will search for an expression"
$Pattern = "\s"
$matches = [regex]::match($strText, $pattern)

"Result of using the match method, we get the following:"
$matches

$strReplace = [regex]::replace($strText, $pattern, "_")
"Now we will replace, using the same pattern. We will use
an underscore to replace the space between words:"

$strReplace
```

Using Command-Line Arguments

Modifying a script at run time is an important time-saving, labor-saving, and flexibility-preserving technique. In many companies, first-level support is given the ability to run scripts but not to create scripts. The first-level support personnel do not have access to script editors, nor are they expected to know how to modify a script at design time. The solution is to use command-line arguments that modify the behavior of the script. In this manner, the scripts become almost like custom-written utilities that are edited by the user, rather than components that are modified via a series of switches and parameters. An example of this technique is shown in the ArgsShare.ps1 script.

The ArgsShare.ps1 script defines a simple function that is used to perform the WMI query. It takes a single argument from the command line when the script is run. This will determine the kind of shares that are returned.

An *if ... else* statement is used to determine if a command-line argument is present. If it is not present, then a friendly help message is displayed that suggests running help for the script. In reality, anything that is not a recognized as a valid argument will result in displaying the help string. The help message suggests the common *question mark* switch.

Once it is determined a valid command-line argument is present, the *switch* statement will assign the appropriate value to the *$strShare* variable, and will then call the WMI function. This procedure allows a user to type in a simple noun such as: *admin*, *print, file, ipc,* or *all* and generate the appropriate WMI query. However, WMI expects a valid share type integer. By using *switch* in this way, you generate the appropriate WMI query based upon input received from the command line. If an unexpected command-line argument is supplied, the default switch is used; this simply prints the help message. You can change this to perform an *all* type of query or some other default WMI query, if desired. You can even paste your default WMI query into the *if(!args)* statement and allow the default query to run when there is no argument present. This mimics the behavior of some Windows command-line utilities. The ArgsShare.ps1 script is shown here.

ArgsShare.ps1

```
Function FunWMI($strShare)
 {
  Get-WmiObject win32_share -Filter "type = $strShare"
 }

if(!$args)
{ "you must supply an argument. Try ArgsShare.ps1 ?"}
ELSE
{
$strShare = $args
switch ($strShare)
  {
   "admin" { $strShare = 2147483648 ; funwmi($strShare) }
   "print" { $strShare = 2147483649 ; funwmi($strShare) }
   "file"  { $strShare = 0 ; funwmi($strShare) }
   "ipc"   { $strShare = 2147483651 ; funwmi($strShare) }
   "all"   { Get-WmiObject win32_share }
   Default { Write-Host "You must supply either: admin, print, file, ipc, or all `n
             Example: > ArgsShare.ps1 admin" }
  }
}
```

Summary

In this chapter, we first examined the scripting policy provided by Windows PowerShell. We looked at the steps involved in configuring Windows PowerShell for scripting use, explored the various flow control statements, and examined scripts that use flow control for advanced scripting needs. We looked at implementing decision making in Windows PowerShell and saw how encapsulated logic can vastly simplify network administration tasks by acting upon routine events when they are presented to the script. Finally, we explored the use of regular expressions to provide advanced pattern-matching capabilities to both scripts and cmdlets.