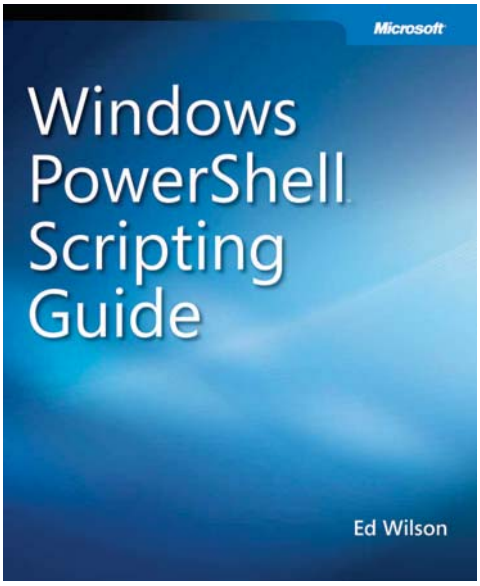

Windows PowerShell™ Scripting Guide

Ed Wilson

PREVIEW CONTENT This excerpt contains uncorrected manuscript from an upcoming Microsoft Press title, for early preview, and is subject to change prior to release. This excerpt is from *Windows PowerShell™ Scripting Guide* from Microsoft Press (ISBN 978-0-7356-2279-1, copyright 2008 Ed Wilson, all rights reserved), and is provided without any express, statutory, or implied warranties

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

Microsoft Press

978-0-7356-2279-1

Table of Contents

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 CD 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 hotfixID 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.

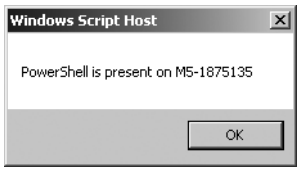

Figure 1-1 The FindPowerShell.vbs script displays a pop-up box indicating that Windows PowerShell has been found.

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 deployment follow:

1. Create a Microsoft Systems Management Server (SMS) package and advertise it to the appropriate organizational unit (OU) or collection.
2. Create a Group Policy Object (GPO) in Active Directory and link it to the appropriate OU.
3. 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 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 the 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.

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 Start-up 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.

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

   ```
   PowerShell -nologo
   ```

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

   ```
   PowerShell -version 1.0
   ```

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

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

4. 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.

Whatlf.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.

Whatlf2.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 has a high level of discoverability. That is, to learn how to use Windows PowerShell you can simply use Windows PowerShell. Online help serves an important role in assisting in this discoverability. 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.

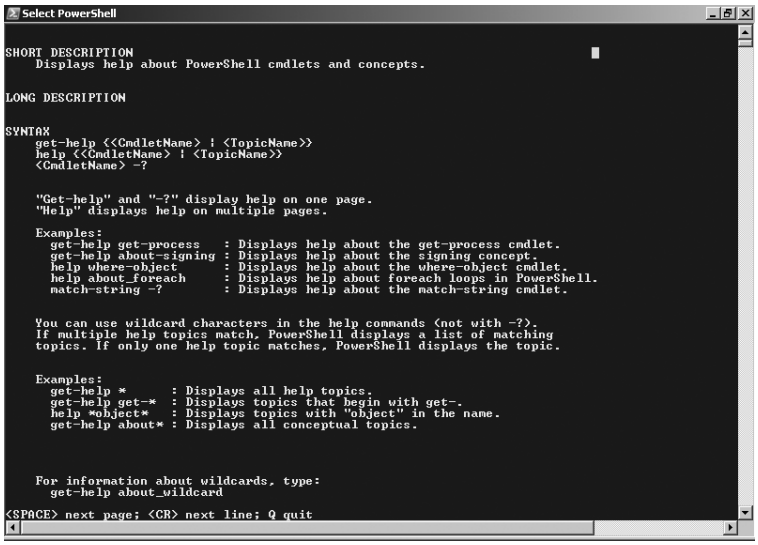

Figure 1-3 By using the *more* function, you can display lengthy help topics one page at a time.

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 neither 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 is a much easier to read output 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
GetAccessControl(), 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. 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. The chapter covered the creation of Windows PowerShell profiles and explored various methods of launching both Windows PowerShell and Windows PowerShell commands. It included extending the features of Windows PowerShell via the creation of custom aliases and functions. The chapter concluded with a discussion of three Windows PowerShell cmdlets: Get-Help, Get-Command, and Get-Member.