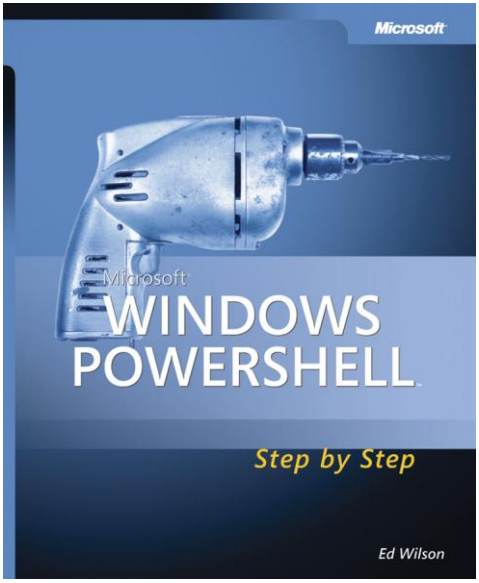

Microsoft® Windows PowerShell™ Step By Step

Ed Wilson

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/10329.aspx

9780735623958
Publication Date: May 2007

Microsoft Press

Chapter 1

Overview of Windows PowerShell

After completing this chapter, you will be able to:

- Understand basic use and capabilities of Microsoft Windows PowerShell
- Install Windows PowerShell
- Use basic command-line utilities inside Windows PowerShell
- Use Windows PowerShell help
- Run basic Windows PowerShell cmdlets
- Get help on basic Windows PowerShell cmdlets
- Configure Windows PowerShell to run scripts

The release of Windows PowerShell marks a significant advance for the Windows network administrator. Combining the power of a full-fledged scripting language, with access to command-line utilities, Windows Management Instrumentation (WMI), and even VBScript, PowerShell provides both the power and ease of use that have been missing from the Windows platform since the beginning of time. All the scripts mentioned in this chapter can be found in the corresponding scripts folder on the CD.

Understanding Windows PowerShell

Perhaps the biggest obstacle for a Windows network administrator in migrating to Windows PowerShell is understanding what the PowerShell actually is. In some respects, it is like a replacement for the venerable CMD (command) shell. As shown here, after the Windows PowerShell is launched, you can use *cd* to change the working directory, and then use *dir* to produce a directory listing in exactly the same way you would perform these tasks from the CMD shell.

```
Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.

PS C:\Documents and Settings\edwilson> cd c:\
PS C:\> dir

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

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----          7/2/2006  12:14 PM            audioBOOK
d----         1/13/2006   9:34 AM            bt
d----         11/4/2006   2:57 AM            Documents and Settings
```

```
d----          2/6/2006   2:49 PM            DsoFile
d----          9/5/2006  11:30 AM            fso
d----         7/21/2006   3:08 AM            fso2
d----        11/15/2006   9:57 AM            OutlookMail
d-r--        11/20/2006   4:44 PM            Program Files
d----         7/16/2005  11:52 AM            RAS
d----         1/30/2006   9:30 AM            smartPhone
d----         11/1/2006  11:35 PM            Temp
d----         8/31/2006   6:48 AM            Utils
d----         1/30/2006   9:10 AM            vb05sbs
d----        11/21/2006   5:36 PM            WINDOWS
-a---         7/16/2005  10:39 AM          0 AUTOEXEC.BAT
-a---         11/7/2006   1:09 PM       3988 bar.emf
--r-s         8/27/2006   6:37 PM        211 boot.ini
-a---         7/16/2005  10:39 AM          0 CONFIG.SYS
-a---         8/16/2006  11:42 AM         60 MASK.txt
-a---          4/5/2006   3:09 AM        288 MRED1.log
-a---         9/28/2006  11:20 PM      16384 mySheet.xls
-a---         9/19/2006   4:28 AM       2974 new.txt
-a---        11/15/2006   2:08 PM       6662 notepad
-a---         9/19/2006   4:23 AM       4887 old.txt
-a---          6/3/2006  11:11 AM        102 Platform.ini

PS C:\>
```

You can also combine "traditional" CMD interpreter commands with some of the newer utilities such as *fsutil*. This is shown here:

```
PS C:\> md c:\test

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

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         11/23/2006  11:42 AM            test

PS C:\> cd c:\test
PS C:\test> fsutil file createNew c:\test\myNewFile.txt 1000
File c:\test\myNewFile.txt is created
PS C:\test> dir

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

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         11/23/2006  11:43 AM       1000 myNewFile.txt

PS C:\test> del *.txt
PS C:\test> cd c:\
PS C:\> rd c:\test
PS C:\>
```

We have been using Windows PowerShell in an interactive manner. This is one of the primary uses of PowerShell and is accomplished by opening a PowerShell prompt and typing commands. The commands can be entered one at a time, or they can be grouped together like a batch file. We will look at this later because you need more information to understand it.

Using Cmdlets

In addition to using traditional programs and commands from the CMD.exe command interpreter, we can also use the commandlets (cmdlets) that are built into PowerShell. Cmdlets are name-created by the Windows PowerShell team to describe the commands that are built into PowerShell. They are like executable programs, but they take advantage of the facilities built into Windows PowerShell, and therefore are easy to write. They are not scripts, which are uncompiled code, because they are built using the services of a special .NET Framework namespace. Windows PowerShell comes with more than 120 cmdlets that are designed to assist the network administrator or consultant to leverage the power of PowerShell without having to learn the PowerShell scripting language. These cmdlets are documented in Appendix A. 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 seen in Appendix B.

Installing Windows PowerShell

It is unfortunate that Windows PowerShell is not installed by default on any of the current Windows operating systems, including Windows Vista. It is installed with Exchange Server 2007 because Exchange leverages Windows PowerShell for management. This is a tremendous advantage to Exchange admins because it means that everything that can be done through the Exchange Admin tool can also be done from a PowerShell script or cmdlet.

Windows PowerShell can be installed on Windows XP SP2, Windows Server 2003 SP1, and Windows Vista. Windows PowerShell requires Microsoft .NET Framework 2.0 (or greater) and will generate the error shown in Figure 1-1 if this level of the .NET Framework is not installed.

Figure 1-1 A Setup error is generated if .NET Framework 2.0 is not present

To prevent frustration during the installation, it makes sense to use a script that checks for the operating system (OS), service pack level, and .NET Framework 2.0. A sample script that will check for the prerequisites is DetectPowerShellRequirements.vbs, which follows.

DetectPowerShellRequirements.vbs

```
strComputer = "."
wmiNS = "\root\cimv2"
wmiQuery = "Select name from win32_Product where name like '%.NET Framework 2.0%'"
wmiQuery1 = "Select * from win32_OperatingSystem"

WScript.Echo "Retrieving settings on " & _ CreateObject("wscript.network").computername
    & " this will take some time ..."
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.ExecQuery(wmiQuery)
Set colItems1= objWMIService.ExecQuery(wmiQuery1,,RtnImmedFwdOnly)

If colItems.count <>1 Then
    WScript.Echo ".NET Framework 2.0 is required for PowerShell"
    Else
    WScript.Echo ".NET Framework 2.0 detected"
End If

For Each objItem1 In colItems1
    osVER= objItem1.version
    osSP= objItem1.ServicePackMajorVersion
Next

Select Case osVER
Case "5.1.2600"
    If osSP < 2 Then
        WScript.Echo "Service Pack 2 is required on Windows XP"
    Else
        WScript.Echo "Service Pack",osSP,"detected on",osVER
    End If
Case "5. 2.3790"
    If osSP <1 Then
        WScript.Echo "Service Pack 1 is required on Windows Server 2003"
    Else
        WScript.Echo "Service Pack",osSP,"detected on",osVER
    End if
Case "XXX"
    WScript.Echo "No service pack is required on Windows Vista"
Case Else
    WScript.Echo "Windows PowerShell does not install on Windows version " & osVER
End Select
```

Deploying Windows PowerShell

After Windows PowerShell is downloaded from *http://www.Microsoft.com/downloads*, you can deploy Windows PowerShell to your enterprise by using any of the standard methods you currently use. A few of the methods some customers have used to accomplish Windows PowerShell deployment are listed next.

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 (AD) and link it to the appropriate OU.

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.

Note To use a command line utility in Windows PowerShell, launch Windows PowerShell by using *Start | Run | PowerShell*. At the PowerShell prompt, type in the command to run.

Using Command Line Utilities

As mentioned earlier, command-line utilities can be used directly within Windows PowerShell. The advantages of using command-line utilities in Windows PowerShell, as opposed to simply running them in the CMD interpreter, are the Windows PowerShell pipelining and formatting features. Additionally, if you have batch files or CMD files that already utilize existing command-line utilities, they can easily be modified to run within the Windows PowerShell environment. This command is in the RunningIpconfigCommands.txt file.

Running *ipconfig* commands

1. Start the Windows PowerShell by using *Start | Run | Windows PowerShell*. The PowerShell prompt will open by default at the root of your Documents And Settings.
2. Enter the command *ipconfig /all*. This is shown here:

```
PS C:\> ipconfig /all
```

3. Pipe the result of *ipconfig /all* to a text file. This is illustrated here:

```
PS C:\> ipconfig /all >ipconfig.txt
```

4. Use Notepad to view the contents of the text file. This is shown here:

```
PS C:\> notepad ipconfig.txt
```

Typing a single command into Windows PowerShell is useful, but at times you may need more than one command to provide troubleshooting information, or configuration details to assist with setup issues or performance problems. This is where Windows PowerShell really shines. In the past, one would have to either write a batch file or type the commands manually.

Note Netdiag.exe referenced in the TroubleShoot.bat file is not part of the standard Windows install, but is a resource kit utility that can be downloaded from *http://www.microsoft.com/downloads*.

This is seen in the TroubleShoot.bat script that follows.

TroubleShoot.bat

```
ipconfig /all >C:\tshoot.txt
route print >>C:\tshoot.txt
netdiag /q >>C:\tshoot.txt
net statistics workstation >>C:\tshoot.txt
```

Of course, if you typed the commands manually, then you had to wait for each command to complete before entering the subsequent command. In that case, it was always possible to lose your place in the command sequence, or to have to wait for the result of each command. The Windows PowerShell eliminates this problem. You can now enter multiple commands on a single line, and then leave the computer or perform other tasks while the computer produces the output. No batch file needs to be written to achieve this capability.

Tip Use multiple commands on a single Windows PowerShell line. Type each complete command, and then use a semicolon to separate each command.

The use of this procedure is seen in the Running multiple commands procedure. The command used in the procedure are in the RunningMultipleCommands.txt file.

Running multiple commands

1. Start the Windows PowerShell by using *Start | Run | Windows PowerShell*. The PowerShell prompt will open by default at the root of your Documents And Settings.
2. Enter the *ipconfig /all* command. Pipe the output to a text file called Tshoot.txt by using the redirection arrow (>). This is the result:

   ```
   ipconfig /all >tshoot.txt
   ```

3. On the same line, use a semicolon to separate the *ipconfig /all* command from the *route print* command. Append the output from the command to a text file called Tshoot.txt by using the redirect and append arrow (>>). The command to this point is shown as follows:

   ```
   ipconfig /all >tshoot.txt; route print >>tshoot.txt
   ```

4. On the same line, use a semicolon to separate the *route print* command from the *netdiag /q* command. Append the output from the command to a text file called Tshoot.txt by using the redirect and append arrow. The command to this point is shown here:

   ```
   ipconfig /all >tshoot.txt; route print >>tshoot.txt; netdiag /q >>tshoot
   .txt
   ```

5. On the same line, use a semicolon to separate the *netdiag /q* command from the *net statistics workstation* command. Append the output from the command to a text file called Tshoot.txt by using the redirect and append arrow. The completed command looks like the following:

```
ipconfig /all >tshoot.txt; route print >>tshoot.txt; netdiag /q >>tshoot
.txt; net statistics workstation >>tshoot.txt
```

Security Issues with Windows PowerShell

As with any tool as versatile as Windows PowerShell, there are bound to be 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 Documents And Settings folder; this ensures you are in a directory where you will have permission to perform certain actions and activities. This is far safer than opening at the root of the drive, or even opening in system root.

To change to a directory, you cannot automatically go up to the next level; you must explicitly name the destination of the change directory operation.

The running of scripts is disabled by default and can be easily managed through group policy.

Controlling Execution of PowerShell Cmdlets

Have you ever opened a CMD interpreter prompt, typed in a command, and pressed Enter so that you could see what it does? What if that command happened to be *Format C:*? Are you sure you want to format your C drive? In this section, we will look at some arguments that can be supplied to cmdlets that allow you to control the way they execute. Although not all cmdlets support these arguments, most of those included with Windows PowerShell do. The three arguments we can use to control execution are -whatif, -confirm, and suspend. Suspend is not really an argument that is supplied to a cmdlet, but rather is an action you can take at a confirmation prompt, and is therefore another method of controlling execution.

Note To use -whatif in a Windows PowerShell prompt, enter the cmdlet. Type the -whatif parameter after the cmdlet.

Most of the Windows PowerShell cmdlets support a "prototype" mode that can be entered using the -whatif parameter. The implementation of -whatif can be decided on by the person developing the cmdlet; however, it is the recommendation of the Windows PowerShell team that developers implement -whatif. The use of the -whatif argument is seen in the procedure below. The commands used in the procedure are in the UsingWhatif.txt file.

Using -whatif to prototype a command

1. Start the Windows PowerShell by using *Start | Run | Windows PowerShell*. The PowerShell prompt will open by default at the root of your Documents And Settings.
2. Start an instance of Notepad.exe. Do this by typing **notepad** and pressing the Enter key. This is shown here:

```
notepad
```

3. Identify the Notepad process you just started by using the *Get-Process* cmdlet. Type enough of the process name to identify it, and then use a wild card asterisk (*) to avoid typing the entire name of the process. This is shown as follows:

```
get-process note*
```

4. Examine the output from the *Get-Process* cmdlet, and identify the process ID. The output on my machine is shown here. Please note that in all likelihood, the process ID used by your instance of Notepad.exe will be different from the one on my machine.

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   1056 notepad
```

5. Use -whatif to see what would happen if you used *Stop-Process* to stop the process ID you obtained in step 4. This process ID will be found under the Id column in your output. Use the -id parameter to identify the Notepad.exe process. The command is as follows:

```
stop-process -id 1056 -whatif
```

6. Examine the output from the command. It tells you that the command will stop the Notepad process with the process ID that you used in your command.

```
What if: Performing operation "Stop-Process" on Target "notepad (1056)"
```

Tip To confirm the execution of a cmdlet, launch Windows PowerShell by using *Start | Run | Windows PowerShell*. At the Windows PowerShell prompt, supply the -whatif argument to the cmdlet.

Confirming Commands

As we saw in the previous section, we can use -whatif to prototype a cmdlet in Windows PowerShell. This is useful for seeing what a command would do; however, if we want to be prompted before the execution of the command, we can use the -confirm argument. The commands used in the Confirming the execution of cmdlets procedure are listed in the ConfirmingExecutionOfCmdlets.txt file.

Confirming the execution of cmdlets

1. Start the Windows PowerShell by using *Start | Run | Windows PowerShell*. The PowerShell prompt will open by default at the root of your Documents And Settings.
2. Start an instance of Notepad.exe. Do this by typing **notepad** and pressing the Enter key. This is shown here:

```
notepad
```

3. Identify the Notepad process you just started by using the *Get-Process* cmdlet. Type enough of the process name to identify it, and then use a wild card asterisk (*) to avoid typing the entire name of the process. This is illustrated here:

```
get-process note*
```

4. Examine the output from the *Get-Process* cmdlet, and identify the process ID. The output on my machine is shown here. Please note that in all likelihood, the process ID used by your instance of Notepad.exe will be different from the one on my machine.

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   1768 notepad
```

5. Use the -confirm argument to force a prompt when using the *Stop-Process* cmdlet to stop the Notepad process identified by the *get-process note** command. This is shown here:

```
stop-process -id 1768 -confirm
```

6. The *Stop-Process* cmdlet, when used with the -confirm argument, displays the following confirmation prompt:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (1768)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

7. Type **y** and press Enter. The Notepad.exe process ends. The Windows PowerShell prompt returns to the default ready for new commands, as shown here:

```
PS C:\>
```

Tip To suspend cmdlet confirmation, at the confirmation prompt from the cmdlet, type **s** and press Enter

Suspending Confirmation of Cmdlets

The ability to prompt for confirmation of the execution of a cmdlet is extremely useful and at times may be vital to assisting in maintaining a high level of system uptime. There are times when you have typed in a long command and then remember that you need to do something else first. For such eventualities, you can tell the confirmation you would like to suspend execution of the command. The commands used for suspending execution of a cmdlet are in the SuspendConfirmationOfCmdlets.txt file.

Suspending execution of a cmdlet

1. Start the Windows PowerShell by using *Start | Run | Windows PowerShell*. The PowerShell prompt will open by default at the root of your Documents And Settings.
2. Start an instance of Notepad.exe. Do this by typing **notepad** and pressing the Enter key. This is shown here:

```
notepad
```

3. Identify the Notepad process you just started by using the *Get-Process* cmdlet. Type enough of the process name to identify it, and then use a wild card asterisk (*) to avoid typing the entire name of the process. This is shown here:

```
get-process note*
```

4. Examine the output from the *Get-Process* cmdlet, and identify the process ID. The output on my machine is seen below. Please note that in all likelihood, the process ID used by our instance of Notepad.exe will be different from the one on my machine.

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   3576 notepad
```

5. Use the -confirm argument to force a prompt when using the *Stop-Process* cmdlet to stop the Notepad process identified by the *Get-Process Note** command. This is illustrated here:

```
stop-process -id 3576 -confirm
```

6. The *Stop-Process* cmdlet, when used with the -confirm argument, displays the following confirmation prompt:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (3576)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

7. To suspend execution of the *Stop-Process* cmdlet, enter **s**. A triple arrow prompt will appear, as follows:

```
PS C:\>>>
```

8. Obtain a list of all the running processes that begin with the letter n. Use the *Get-Process* cmdlet to do this. The syntax is as follows:

```
get-process n*
```

9. On my machine, two processes appear. The Notepad process we launched earlier, and another process. This is shown here:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   3576 notepad
     75       2     1776       2708    23     0.09    632 nvsvc32
```

10. Return to the previous confirmation prompt by typing **exit**. This is shown here:

```
exit
```

11. Once again, the confirmation prompt appears as follows:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (3576)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

12. Type **y** and press Enter to stop the Notepad process. There is no further confirmation. The prompt will now display the default Windows PowerShell PS>, as shown here:

```
PS C:\>
```

Working with Windows PowerShell

Windows PowerShell can be used as a replacement for the CMD interpreter. Its many built-in cmdlets allow for large number of activities. These cmdlets can be used in a stand-alone fashion, or they can be run together as a group.

Accessing Windows PowerShell

After Windows PowerShell is installed, it becomes available for immediate use. However, using the Windows flag key on the keyboard and pressing the letter **r** to bring up a *run* command prompt, or "mousing around" and and using *Start | Run | Windows 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 was so useful, as a matter of fact, that I wrote a script to do this. This script can be called through a logon script to automatically deploy the shortcut on the desktop. The script is called CreateShortCutToPowerShell.vbs, and is as follows:

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
```

Configuring Windows PowerShell

Many items can be configured for Windows PowerShell. 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 has 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>
```

Controlling PowerShell launch options

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

   ```
   PowerShell -nologo
   ```

2. Launch a specific version of Windows PowerShell by using the -version argument. This is shown here:

   ```
   PowerShell -version 1
   ```

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

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

4. Launch Windows PowerShell, execute a specific command, and then exit by using the -command argument. The command itself must be prefixed by the ampersand sign (&) and enclosed in curly brackets. This is shown here:

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

Supplying Options for Cmdlets

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

Table 1-1 Common Parameters

Parameter	Meaning
-whatif	Tells the cmdlet to not execute but to tell you what would happen if the cmdlet were to run
-confirm	Tells the cmdlet to prompt before 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. Allowed 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 before calling the next cmdlet in the pipeline

Note To get help on any cmdlet, use the *Get-Help cmdletname* cmdlet.

Working with the Help Options

Windows PowerShell has a high level of discoverability; that is, to learn how to use PowerShell, you can simply use 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 follows:

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

This command prints out help about the *Get-Help* cmdlet. The output from this cmdlet is illustrated 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 a
    nd 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 good thing about online help with the Windows PowerShell is that it not only displays help about commands, which you would expect, but also has three 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 follows:

```
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 wild card, such as an asterisk (*), to obtain the name of the cmdlet. This is shown here:

```
get-help get*
```

This technique of using a wild card operator can be extended further. If you remember that the cmdlet was a *get* cmdlet, and that it started with the letter p, you can 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 cannot exactly remember the syntax. For this scenario, you can use the -examples argument. For example, for the *Get-PSDrive* cmdlet, you would use *Get-Help* with the -examples argument, as follows:

```
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 formatted output is shown in Figure 1-2.

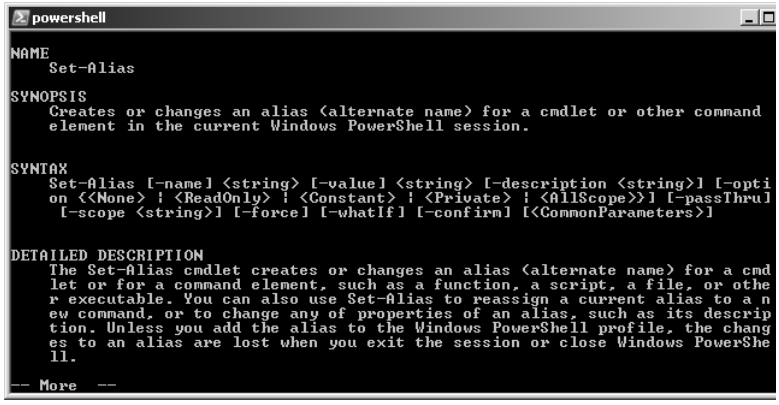

Figure 1-2 Using help to display information one page at a time

Getting tired of typing *Get-Help* all the time? 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. The commands used for this are in the CreateAliasToGet-Help.txt file. An alias is a shortcut key stroke combination that will launch a program or cmdlet when typed. In the creating an alias for the *Get-Help* cmdlet procedure, we will assign the *Get-Help* cmdlet to the gh key combination.

Note To create an alias for a cmdlet, confirm there is not already an alias to the cmdlet by using *Get-Alias*. Use *Set-Alias* to assign the cmdlet to a unique key stroke combination.

Creating an alias for the *Get-Help* cmdlet

1. Start Windows PowerShell by using *Start* | *Run* | *Windows PowerShell*. The PowerShell prompt will open by default at the root of your Documents And Settings.
2. Retrieve an alphabetic listing of all currently defined aliases, and inspect the list for one assigned to either the *Get-Help* cmdlet or the key stroke combination gh. The command to do this is as follows:

```
get-alias |sort
```

3. After you have determined that there is no alias for the *Get-Help* cmdlet, and that none is assigned to the gh key stroke 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
```

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

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

Exploring Commands: Step-by-Step Exercises

In this exercise, we explore the use of command-line utilities in Windows PowerShell. You will see that it is as easy to use command-line utilities in the Windows PowerShell as in the CMD interpreter; however, by using such commands in the Windows PowerShell, you gain access to new levels of functionality.

1. Start Windows PowerShell by using *Start* | *Run* | *Windows PowerShell*. The PowerShell prompt will open by default at the root of your Documents And Settings.
2. Change to the C:\ root directory by typing **cd C:** inside the PowerShell prompt:

```
Cd c:\
```

3. Obtain a listing of all the files in the C:\ root directory by using the *dir* command:

```
dir
```

4. Create a directory off the C:\ root directory by using the *md* command:

```
Md mytest
```

5. Obtain a listing of all files and folders off the root that begin with the letter m:

```
Dir m*
```

6. Change the working directory to the PowerShell working directory. You can do this by using the *Set-Location* command as follows:

```
Set-location $pshome
```

7. Obtain a listing of memory counters related to the available bytes by using the *typeperf* command. This command is shown here:

```
typeperf "\memory\available bytes"
```

8. After a few counters have been displayed in the PowerShell window, use the *ctrl-c* command to break the listing.

9. Display the current boot configuration by using the *bootcfg* command:

```
Bootcfg
```

10. Change the working directory back to the C:\Mytest directory you created earlier:

```
set-location c:\mytest
```

11. Create a file named Mytestfile.txt in the C:\Mytest directory. Use the *fsutil* utility, and make the file 1,000 bytes in size. To do this, use the following command:

```
fsutil file createnew mytestfile.txt 1000
```

12. Obtain a "directory listing" of all the files in the C:\Mytest directory by using the *Get-ChildItem* cmdlet. This is shown here:

```
get-childitem
```

13. Print out the current date by using the *Get-Date* cmdlet. This is shown here:

```
get-date
```

14. Clear the screen by using the *cls* command. This is shown here:

```
cls
```

15. Print out a listing of all the cmdlets built into Windows PowerShell. To do this, use the *Get-Command* cmdlet. This is shown here:

```
get-command
```

16. Use the *Get-Command* cmdlet to get the *Get-Alias* cmdlet. To do this, use the -name argument while supplying *Get-Alias* as the value for the argument. This is shown here:

```
get-command -name get-alias
```

17. This concludes the step-by-step exercise. Exit the Windows PowerShell by typing **exit** and pressing Enter.

One Step Further: Obtaining Help

In this exercise, we use various help options to obtain assistance with various cmdlets.

1. Start Windows PowerShell by using *Start | Run | Windows PowerShell*. The PowerShell prompt will open by default at the root of your Documents And Settings.
2. Use the *Get-Help* cmdlet to obtain help about the *Get-Help* cmdlet. Use the command *Get-Help Get-Help* as follows:

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

3. To obtain detailed help about the *Get-Help* cmdlet, use the -detailed argument as follows:

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

4. To retrieve technical information about the *Get-Help* cmdlet, use the -full argument. This is shown here:

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

5. If you only want to obtain a listing of examples of command usage, use the -examples argument as follows:

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

6. Obtain a listing of all the informational help topics by using the *Get-Help* cmdlet and the about noun with the asterisk (*) wild card operator. The code to do this is shown here:

```
get-help about*
```

7. Obtain a listing of all the help topics related to *get* cmdlets. To do this, use the *Get-Help* cmdlet, and specify the word "get" followed by the wild card operator as follows:

```
get-help get*
```

8. Obtain a listing of all the help topics related to *set* cmdlets. To do this, use the *Get-Help* cmdlet followed by the "set" verb followed by the asterisk wild card. This is shown here:

```
get-help set*
```

9. This concludes the one step further exercise. Exit the Windows PowerShell by typing **exit** and pressing Enter.

Chapter 1 Quick Reference

To	Do This
Use an external command-line utility	Type the name of the command-line utility while inside Windows PowerShell
Use multiple external command-line utilities sequentially	Separate each command-line utility with a semicolon on a single Windows PowerShell line
Obtain a list of running processes	Use the *Get-Process* cmdlet
Stop a process	Use the *Stop-Process* cmdlet and specify either the name or the process ID as an argument
Model the effect of a cmdlet before actually performing the requested action	Use the -whatif argument
Instruct Windows PowerShell to startup, run a cmdlet, and then exit	Use the *PowerShell* command while prefixing the cmdlet with the ampersand sign and enclosing the name of the cmdlet in curly brackets
Prompt for confirmation before stopping a process	Use the *Stop-Process* cmdlet while specifying the -confirm argument