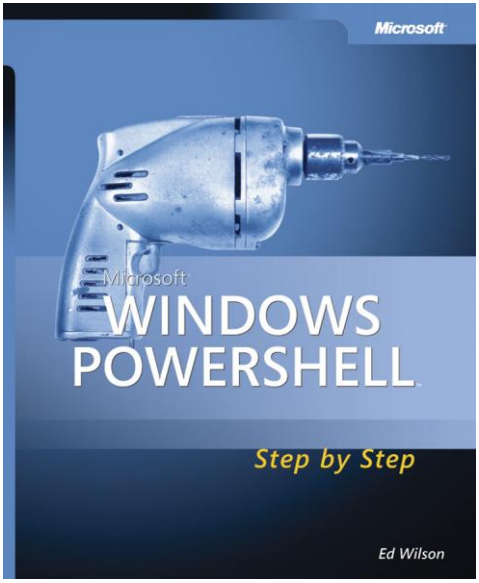

Microsoft® Windows PowerShell™ Step By Step

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

Table of Contents

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.

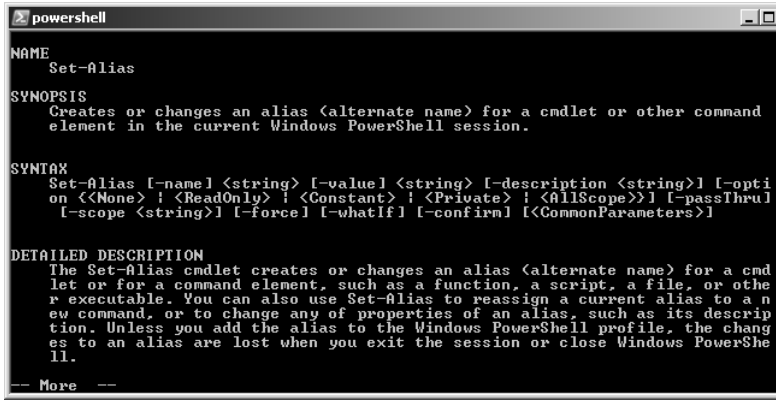

Figure 1-2 Using help to display information one page at a time

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

Chapter 2

Using Windows PowerShell Cmdlets

After completing this chapter, you will be able to:

- Understand the basic use of Microsoft Windows PowerShell cmdlets
- Use *Get-Command* to retrieve a listing of cmdlets
- Configure search options
- Configure output parameters
- Use *Get-Member*
- Use *New-Object*

The inclusion of a large amount of cmdlets in Windows PowerShell makes it immediately useful to network administrators and others who need to perform various maintenance and administrative tasks on their Windows servers and desktop systems. In this chapter, we review several of the more useful cmdlets as a means of highlighting the power and flexibility of Windows PowerShell. However, the real benefit of this chapter is the methodology we use to discover the use of the various cmdlets. All the scripts mentioned in this chapter can be found in the corresponding scripts folder on the CD.

Understanding the Basics of Cmdlets

In Chapter 1, Overview of Windows PowerShell, we learned about using the various help utilities available that demonstrate how to use cmdlets. We looked at a couple of cmdlets that are helpful in finding out what commands are available and how to obtain information about them. In this section, we describe some additional ways to use cmdlets in Windows PowerShell.

Tip Typing long cmdlet names can be somewhat tedious. To simplify this process, type enough of the cmdlet name to uniquely distinguish it, and then press the Tab key on the keyboard. What is the result? *Tab Completion* completes the cmdlet name for you. This also works with argument names and other things you are entering. Feel free to experiment with this great time-saving technique. You may never have to type **get-command** again!

Because the cmdlets return objects instead of "string values," we can obtain additional information about the returned objects. The additional information would not be available to us if

we were working with just string data. To do this, we can use the pipe character (|) to take information from one cmdlet and feed it to another cmdlet. This may seem complicated, but it is actually quite simple and, by the end of this chapter, will seem quite natural. At the most basic level, consider obtaining a directory listing; after you have the directory listing, perhaps you would like to format the way it is displayed—as a table or a list. As you can see, these are two separate operations: obtaining the directory listing, and formatting the list. The second task will take place on the right side of the pipe.

Using the *Get-ChildItem* Cmdlet

In Chapter 1, we 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*.

Just the Steps **Obtaining a directory listing** In a Windows PowerShell prompt, enter the *Get-ChildItem* cmdlet followed by the directory to list. Example:

```
get-childitem C:\
```

In Windows PowerShell, there actually is no cmdlet called *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 in the CMD.exe interpreter. The alias *dir* is used when we use the *Get-Alias* cmdlet to resolve the association, as follows:

```
PS C:\> get-alias dir

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

If you use the *Get-ChildItem* cmdlet to obtain the directory listing, it will obtain a listing the same as *dir* because *dir* is simply an alias for *Get-ChildItem*. This is shown here:

```
PS C:\> get-childitem C:\

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

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----          7/2/2006   3:14 PM            audioBOOK
d----         11/4/2006   4:57 AM            Documents and Settings
d----          2/6/2006   4:49 PM            DsoFile
d----          9/5/2006   2:30 PM            fso
d----        11/30/2006   2:08 PM            fso1
d----         7/21/2006   6:08 AM            fso2
d----         12/2/2005   5:41 AM            German
d----         9/24/2006   1:54 AM            music
d----        12/10/2006   6:54 AM            mytest
d----        12/13/2006   8:30 AM            OutlookMail
```

```
d-r--        11/20/2006   6:44 PM            Program Files
d----         7/16/2005   2:52 PM            RAS
d----         1/30/2006  11:30 AM            smartPhone
d----         11/2/2006   1:35 AM            Temp
d----         8/31/2006   9:48 AM            Utils
d----         1/30/2006  11:10 AM            vb05sbs
d----         12/5/2006   8:01 AM            WINDOWS
-a---         12/8/2006   7:24 PM      22950 a.txt
-a---         12/5/2006   8:48 AM      23902 alias.txt
-a---         7/16/2005   1:39 PM          0 AUTOEXEC.BAT
-a---         11/7/2006   3:09 PM       3988 bar.emf
--r-s         8/27/2006   9:37 PM        211 boot.ini
-a---         12/3/2006   7:36 AM      21228 cmdlets.txt
-a---        12/13/2006   9:44 AM     273612 commandHelp.txt
-a---        12/10/2006   7:34 AM      21228 commands.txt
-a---         7/16/2005   1:39 PM          0 CONFIG.SYS
-a---         12/7/2006   3:14 PM       8261 mySheet.xls
-a---         12/7/2006   5:29 PM       2960 NetDiag.log
-a---         12/5/2006   8:29 AM      16386 notepad
-a---          6/3/2006   2:11 PM        102 Platform.ini
-a---         12/7/2006   5:29 PM      10670 tshoot.txt
-a---         12/4/2006   9:09 PM      52124 VistaResKitScripts.txt
```

If you were to use *Get-Help* and then *dir*, you would receive the same output as if you were to use *Get-Help Get-ChildItem*. In Windows PowerShell, the two can be used in exactly the same fashion.

Just the Steps **Formatting a directory listing using *Format-List*** In a Windows PowerShell prompt, enter the *Get-ChildItem* cmdlet followed by the directory to list followed by the pipe character and the *Format-List* cmdlet. Example:

```
get-childitem C:\ | format-list
```

Formatting output with the *Format-List* 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. Use the *Get-ChildItem* cmdlet to obtain a directory listing of the C:\ directory.

   ```
   get-childItem C:\
   ```

3. Use the *Format-List* cmdlet to arrange the output of *Get-ChildItem*.

   ```
   get-childitem |format-list
   ```

4. Use the -property argument of the *Format-List* cmdlet to retrieve only a listing of the name of each file in the root.

   ```
   get-childitem C:\ | format-list -property name
   ```

5. Use the property argument of the *Format-List* cmdlet to retrieve only a listing of the name and length of each file in the root.

   ```
   get-childitem C:\ | format-list -property name, length
   ```

Using the *Format-Wide* Cmdlet

In the same way that we use the *Format-List* cmdlet to produce an output in a list, we can use the *Format-Wide* cmdlet to produce a more compact output.

Just the Steps **Formatting a directory listing using *Format-Wide*** In a Windows PowerShell prompt, enter the *Get-ChildItem* cmdlet followed by the directory to list followed by the pipe character and the *Format-Wide* cmdlet. Example:

```
get-childitem C:\ | format-wide
```

Formatting output with the *Format-Wide* 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. Use the *Get-ChildItem* cmdlet to obtain a directory listing of the C:\Windows directory.

   ```
   get-childitem C:\Windows
   ```

3. Use the -recursive argument to cause the *Get-ChildItem* cmdlet to walk through a nested directory structure, including only .txt files in the output.

   ```
   get-childitem C:\Windows -recurse -include *.txt
   ```

4. A partial output from the command is shown here:

   ```
       Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Driver Cache

   Mode                 LastWriteTime     Length Name
   ----                 -------------     ------ ----
   -a---         11/26/2004   6:29 AM      13512 yk51x86.txt

       Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Help\Tours\mmTo
       ur

   Mode                 LastWriteTime     Length Name
   ----                 -------------     ------ ----
   -a---          8/4/2004   8:00 AM        807 intro.txt
   -a---          8/4/2004   8:00 AM        407 nav.txt
   -a---          8/4/2004   8:00 AM        747 segment1.txt
   -a---          8/4/2004   8:00 AM        772 segment2.txt
   -a---          8/4/2004   8:00 AM        717 segment3.txt
   -a---          8/4/2004   8:00 AM        633 segment4.txt
   -a---          8/4/2004   8:00 AM        799 segment5.txt
   ```

5. Use the *Format-Wide* cmdlet to adjust the output from the *Get-ChildItem* cmdlet. Use the -columns argument and supply a parameter of 3 to it. This is shown here:

   ```
   get-childitem C:\Windows -recurse -include *.txt |format-wide -column 3
   ```

6. Once this command is run, you will see an output similar to this:

```
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Driver Cache

yk51x86.txt

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Help\Tours\mmTo
    ur

intro.txt                  nav.txt                    segment1.txt
segment2.txt               segment3.txt               segment4.txt
segment5.txt

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Microsoft.NET\F
    ramework\v1.1.4322\1033

SetupENU1.txt              SetupENU2.txt

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Microsoft.NET\F
    ramework\v2.0.50727\Microsoft .NET Framework 2.0

eula.1025.txt              eula.1028.txt              eula.1029.txt
eula.1030.txt              eula.1031.txt              eula.1032.txt
eula.1033.txt              eula.1035.txt              eula.1036.txt
eula.1037.txt              eula.1038.txt              eula.1040.txt
eula.1041.txt              eula.1042.txt              eula.1043.txt
eula.1044.txt              eula.1045.txt              eula.1046.txt
eula.1049.txt              eula.1053.txt              eula.1055.txt
eula.2052.txt              eula.2070.txt              eula.3076.txt
eula.3082.txt
```

7. Use the *Format-Wide* cmdlet to adjust the output from the *Get-ChildItem* cmdlet. Use the property argument to specify the name property, and group the outputs by size. The command shown here appears on two lines; however, when typed into Windows PowerShell, it is a single command and needs to be on the same line:

```
get-childitem C:\Windows -recurse -include *.txt |format-wide -property
name -groupby length -column 3
```

8. A partial output is shown here. Note that although three columns were specified, if there are not three files of the same length, only one column will be used:

```
   Length: 13512

yk51x86.txt

   Length: 807

intro.txt

   Length: 407

nav.txt

   Length: 747

segment1.txt
```

Just the Steps **Formatting a directory listing using *Format-Table*** In a Windows PowerShell prompt, enter the *Get-ChildItem* cmdlet followed by the directory to list followed by the pipe character and the *Format-Table* cmdlet. Example:

```
get-childitem C:\ | format-table
```

Formatting output with the *Format-Table* 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. Use the *Get-ChildItem* cmdlet to obtain a directory listing of the C:\Windows directory

   ```
   get-childitem C:\Windows
   ```

3. Use the -recursive argument to cause the *Get-ChildItem* cmdlet to walk through a nested directory structure, include only .txt files in the output.

   ```
   get-childitem C:\Windows -recurse -include *.txt
   ```

4. Use the *Format-Table* cmdlet to adjust the output from the *Get-ChildItem* cmdlet. This is shown here:

   ```
   get-childitem C:\Windows -recurse -include *.txt |format-table
   ```

5. The command results in the creation of a table, as follows:

```
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Driver Cache

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         11/26/2004   6:29 AM      13512 yk51x86.txt

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Help\Tours\mmTo
    ur

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          8/4/2004   8:00 AM        807 intro.txt
-a---          8/4/2004   8:00 AM        407 nav.txt
-a---          8/4/2004   8:00 AM        747 segment1.txt
-a---          8/4/2004   8:00 AM        772 segment2.txt
-a---          8/4/2004   8:00 AM        717 segment3.txt
-a---          8/4/2004   8:00 AM        633 segment4.txt
-a---          8/4/2004   8:00 AM        799 segment5.txt

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\Microsoft.NET\F
    ramework\v1.1.4322\1033

 Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          3/6/2002   2:36 PM         38 SetupENU1.txt
-a---          3/6/2002   2:36 PM         38 SetupENU2.txt
```

6. Use the -property argument of the *Format-Table* cmdlet and choose the name, length, and last-write-time properties. This is shown here:

```
get-childitem C:\Windows -recurse -include *.txt |format-table -property
name, length, lastwritetime
```

7. This command results in producing a table with the name, length, and last write time as column headers. A sample of this output is shown here:

```
Name                                                Length LastWriteTime
----                                                ------ -------------
yk51x86.txt                                          13512 11/26/2004 6:29:00 AM
intro.txt                                              807 8/4/2004 8:00:00 AM
nav.txt                                                407 8/4/2004 8:00:00 AM
segment1.txt                                           747 8/4/2004 8:00:00 AM
segment2.txt                                           772 8/4/2004 8:00:00 AM
segment3.txt                                           717 8/4/2004 8:00:00 AM
segment4.txt                                           633 8/4/2004 8:00:00 AM
```

Leveraging the Power of *Get-Command*

Using the *Get-Command* cmdlet, you can obtain a listing of all the cmdlets installed on the Windows PowerShell, but there is much more that can be done using this extremely versatile cmdlet. One such method of using the *Get-Command* cmdlet is to use wild card characters. This is shown in the following procedure:

Just the Steps **Searching for cmdlets using wild card characters** In a Windows PowerShell prompt, enter the *Get-Command* cmdlet followed by a wild card character. Example:

```
get-command *
```

Finding commands by using the *Get-Command* 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. Use an alias to refer to the *Get-Command* cmdlet. To find the correct alias, use the *Get-Alias* cmdlet as follows:

```
get-alias g*
```

3. This command produces a listing of all the aliases defined that begin with the letter g. An example of the output of this command is shown here:

```
CommandType     Name                                Definition
-----------     ----                                ----------
Alias           gal                                 Get-Alias
Alias           gc                                  Get-Content
Alias           gci                                 Get-ChildItem
Alias           gcm                                 Get-Command
Alias           gdr                                 Get-PSDrive
```

```
Alias           ghy                             Get-History
Alias           gi                              Get-Item
Alias           gl                              Get-Location
Alias           gm                              Get-Member
Alias           gp                              Get-ItemProperty
Alias           gps                             Get-Process
Alias           group                           Group-Object
Alias           gsv                             Get-Service
Alias           gsnp                            Get-PSSnapin
Alias           gu                              Get-Unique
Alias           gv                              Get-Variable
Alias           gwmi                            Get-WmiObject
Alias           gh                              Get-Help
```

4. Using the *gcm* alias, use the *Get-Command* cmdlet to return the *Get-Command* cmdlet. This is shown here:

```
gcm get-command
```

5. This command returns the *Get-Command* cmdlet. The output is shown here:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Cmdlet          Get-Command                     Get-Command [[-ArgumentList]...
```

6. Using the *gcm* alias to get the *Get-Command* cmdlet, pipe the output to the *Format-List* cmdlet. Use the wild card asterisk (*) to obtain a listing of all the properties of the *Get-Command* cmdlet. This is shown here:

```
gcm get-command |format-list *
```

7. This command will return all the properties from the *Get-Command* cmdlet. The output is shown here:

```
DLL              : C:\WINDOWS\assembly\GAC_MSIL\System.Management.Automation\1.
                   0.0.0__31bf3856ad364e35\System.Management.Automation.dll
Verb             : Get
Noun             : Command
HelpFile         : System.Management.Automation.dll-Help.xml
PSSnapIn         : Microsoft.PowerShell.Core
ImplementingType : Microsoft.PowerShell.Commands.GetCommandCommand
ParameterSets    : {CmdletSet, AllCommandSet}
Definition       : Get-Command [[-ArgumentList] <Object[]>] [-Verb <String[]>]
                   [-Noun <String[]>] [-PSSnapin <String[]>] [-TotalCount <Int3
                   2>] [-Syntax] [-Verbose] [-Debug] [-ErrorAction <ActionPrefe
                   rence>] [-ErrorVariable <String>] [-OutVariable <String>] [-
                   OutBuffer <Int32>]
                   Get-Command [[-Name] <String[]>] [[-ArgumentList] <Object[]>
                   ] [-CommandType <CommandTypes>] [-TotalCount <Int32>] [-Synt
                   ax] [-Verbose] [-Debug] [-ErrorAction <ActionPreference>] [-
                   ErrorVariable <String>] [-OutVariable <String>] [-OutBuffer
                   <Int32>]

Name             : Get-Command
CommandType      : Cmdlet
```

8. Using the *gcm* alias and the *Get-Command* cmdlet, pipe the output to the *Format-List* cmdlet. Use the -property argument, and specify the definition property of the *Get-Command* cmdlet. Rather than retyping the entire command, use the up arrow on your keyboard to retrieve the previous *gcm Get-Command | Format-List** command. Use the Backspace key to remove the asterisk and then simply add -property definition to your command. This is shown here:

```
gcm get-command | format-list -property definition
```

9. This command only returns the property definition for the *Get-Command* cmdlet. The returned definition is shown here:

```
Definition : Get-Command [[-ArgumentList] <Object[]>] [-Verb <String[]>] [-Noun
              <String[]>] [-PSSnapin <String[]>] [-TotalCount <Int32>] [-Syntax
             ] [-Verbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVar
             iable <String>] [-OutVariable <String>] [-OutBuffer <Int32>]
             Get-Command [[-Name] <String[]>] [[-ArgumentList] <Object[]>] [-Co
             mmandType <CommandTypes>] [-TotalCount <Int32>] [-Syntax] [-Verbos
             e] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <Str
             ing>] [-OutVariable <String>] [-OutBuffer <Int32>]
```

10. Because objects are returned from cmdlets instead of simply string data, we can also retrieve the definition of the *Get-Command* cmdlet by directly using the definition property. This is done by putting the expression inside parentheses, and using a "dotted notation," as shown here:

```
(gcm get-command).definition
```

11. The definition returned from the previous command is virtually identical to the one returned by using *Format-List* cmdlet.

12. Use the *gcm* alias and specify the -verb argument. Use *se** for the verb. This is shown here:

```
gcm -verb se*
```

13. The previous command returns a listing of all the cmdlets that contain a verb beginning with se. The result is as follows:

```
CommandType     Name                              Definition
-----------     ----                              ----------
Cmdlet          Select-Object                     Select-Object [[-Property] <...
Cmdlet          Select-String                     Select-String [-Pattern] <St...
Cmdlet          Set-Acl                           Set-Acl [-Path] <String[]> [...
Cmdlet          Set-Alias                         Set-Alias [-Name] <String> [...
Cmdlet          Set-AuthenticodeSignature         Set-AuthenticodeSignature [-...
Cmdlet          Set-Content                       Set-Content [-Path] <String[...
Cmdlet          Set-Date                          Set-Date [-Date] <DateTime> ...
Cmdlet          Set-ExecutionPolicy               Set-ExecutionPolicy [-Execut...
Cmdlet          Set-Item                          Set-Item [-Path] <String[]> ...
Cmdlet          Set-ItemProperty                  Set-ItemProperty [-Path] <St...
Cmdlet          Set-Location                      Set-Location [[-Path] <Strin...
Cmdlet          Set-PSDebug                       Set-PSDebug [-Trace <Int32>]...
Cmdlet          Set-Service                       Set-Service [-Name] <String>...
Cmdlet          Set-TraceSource                   Set-TraceSource [-Name] <Str...
Cmdlet          Set-Variable                      Set-Variable [-Name] <String...
```

14. Use the *gcm* alias and specify the -noun argument. Use *o** for the noun. This is shown here:

```
gcm -noun o*
```

15. The previous command will return all the cmdlets that contain a noun that begins with the letter o. This result is as follows:

```
CommandType     Name                                Definition
-----------     ----                                ----------
Cmdlet          Compare-Object                      Compare-Object [-ReferenceOb...
Cmdlet          ForEach-Object                      ForEach-Object [-Process] <S...
Cmdlet          Group-Object                        Group-Object [[-Property] <O...
Cmdlet          Measure-Object                      Measure-Object [[-Property] ...
Cmdlet          New-Object                          New-Object [-TypeName] <Stri...
Cmdlet          Select-Object                       Select-Object [[-Property] <...
Cmdlet          Sort-Object                         Sort-Object [[-Property] <Ob...
Cmdlet          Tee-Object                          Tee-Object [-FilePath] <Stri...
Cmdlet          Where-Object                        Where-Object [-FilterScript]...
Cmdlet          Write-Output                        Write-Output [-InputObject] ...
```

16. Retrieve only the syntax of the *Get-Command* cmdlet by specifying the -syntax argument. Use the *gcm* alias to do this, as shown here:

```
gcm -syntax get-command
```

17. The syntax of the *Get-Command* cmdlet is returned by the previous command. The output is as follows:

```
Get-Command [[-ArgumentList] <Object[]>] [-Verb <String[]>] [-Noun <String[]>]
[-PSSnapin <String[]>] [-TotalCount <Int32>] [-Syntax] [-Verbose] [-Debug] [-Er
rorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable <String>]
[-OutBuffer <Int32>]
Get-Command [[-Name] <String[]>] [[-ArgumentList] <Object[]>] [-CommandType <Co
mmandTypes>] [-TotalCount <Int32>] [-Syntax] [-Verbose] [-Debug] [-ErrorAction
<ActionPreference>] [-ErrorVariable <String>] [-OutVariable <String>] [-OutBuff
er <Int32>]
```

18. Try to use only aliases to repeat the *Get-Command* syntax command to retrieve the syntax of the *Get-Command* cmdlet. This is shown here:

```
gcm -syntax gcm
```

19. The result of this command is the not the nice syntax description of the previous command. The rather disappointing result is as follows:

```
Get-Command
```

20. This concludes the procedure for finding commands by using the *Get-Command* cmdlet.

Quick Check

Q. To retrieve a definition of the *Get-Command* cmdlet, using the dotted notation, what command would you use?

A. *(gcm get-command).definition*

Using the *Get-Member* Cmdlet

The *Get-Member* cmdlet retrieves information about the members of objects. Although this may not seem very exciting, remember that because everything returned from a cmdlet is an object, we can use the *Get-Member* cmdlet to examine the methods and properties of objects. When the *Get-Member* cmdlet is used with *Get-ChildItem* on the filesystem, it returns a listing of all the methods and properties available to work with the filesystem object.

Objects, Properties, and Methods

One of the more interesting features of Windows PowerShell is that cmdlets return objects. An object is a thing that gives us the ability to either describe something or do something. If we are not going to describe or do something, then there is no reason to create the object. Depending on the circumstances, we may be more interested in the methods, or the properties. As an example, let's consider rental cars. I travel a great deal in my role as a consultant at Microsoft, and I often need to obtain a rental car.

When I get to the airport, I go to the rental car counter, and I use the *New-Object* cmdlet to create the rentalCAR object. When I use this cmdlet, I am only interest in the methods available from the rentalCAR object. I will need to use the DriveDowntheRoad method, the StopAtaRedLight method, and perhaps the PlayNiceMusic method. I am not, however, interested in the properties of the rentalCAR object.

At home, I have a cute little sports car. It has exactly the same methods as the rentalCAR object, but I created the sportsCAR object primarily because of its properties. It is green and has alloy rims, a convertible top, and a 3.5-liter engine. Interestingly enough, it has exactly the same methods as the rentalCAR object. It also has the DriveDowntheRoad method, the StopAtaRedLight method, and the PlayNiceMusic method, but the deciding factor in creating the sportsCAR object was the properties, not the methods.

Just the Steps **Using the *Get-Member* cmdlet to examine properties and methods** In a Windows PowerShell prompt, enter the *Get-ChildItem* cmdlet followed by the path to a folder and pipe it to the *Get-Member* cmdlet. Example:

```
get-childitem C:\ | get-member
```

Using the *Get-Member* 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. Use an alias to refer to the *Get-Alias* cmdlet. To find the correct alias, use the *Get-Alias* cmdlet as follows:

```
get-alias g*
```

3. After you have retrieved the alias for the *Get-Alias* cmdlet, use it to find the alias for the *Get-Member* cmdlet. One way to do this is to use the following command, simply using *gal* in place of the *Get-Alias* name you used in the previous command:

```
gal g*
```

4. The listing of aliases defined that begin with the letter g appears as a result of the previous command. The output is shown here:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Alias           gal                             Get-Alias
Alias           gc                              Get-Content
Alias           gci                             Get-ChildItem
Alias           gcm                             Get-Command
Alias           gdr                             Get-PSDrive
Alias           ghy                             Get-History
Alias           gi                              Get-Item
Alias           gl                              Get-Location
Alias           gm                              Get-Member
Alias           gp                              Get-ItemProperty
Alias           gps                             Get-Process
Alias           group                           Group-Object
Alias           gsv                             Get-Service
Alias           gsnp                            Get-PSSnapin
Alias           gu                              Get-Unique
Alias           gv                              Get-Variable
Alias           gwmi                            Get-WmiObject
Alias           gh                              Get-Help
```

5. Use the *gal* alias to obtain a listing of all aliases that begin with the letter g. Pipe the results to the *Sort-Object* cmdlet, and sort on the property attribute called *definition*. This is shown here:

```
gal g* |sort-object -property definition
```

6. The listings of cmdlets that begin with the letter g are now sorted, and the results of the command are as follows:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Alias           gal                             Get-Alias
Alias           gci                             Get-ChildItem
Alias           gcm                             Get-Command
Alias           gc                              Get-Content
Alias           gh                              Get-Help
```

```
Alias           ghy                            Get-History
Alias           gi                             Get-Item
Alias           gp                             Get-ItemProperty
Alias           gl                             Get-Location
Alias           gm                             Get-Member
Alias           gps                            Get-Process
Alias           gdr                            Get-PSDrive
Alias           gsnp                           Get-PSSnapin
Alias           gsv                            Get-Service
Alias           gu                             Get-Unique
Alias           gv                             Get-Variable
Alias           gwmi                           Get-WmiObject
Alias           group                          Group-Object
```

7. Use the alias for the *Get-ChildItem* cmdlet and pipe the output to the alias for the *Get-Member* cmdlet. This is shown here:

```
gci | gm
```

8. To only see properties available for the *Get-ChildItem* cmdlet, use the membertype argument and supply a value of property. Use *Tab Completion* this time, rather than the *gci | gm* alias. This is shown here:

```
get-childitem | get-member -membertype property
```

9. The output from this command is shown here:

```
   TypeName: System.IO.DirectoryInfo

Name              MemberType Definition
----              ---------- ----------
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;}

   TypeName: System.IO.FileInfo

Name              MemberType Definition
----              ---------- ----------
Attributes        Property   System.IO.FileAttributes Attributes {get;set;}
CreationTime      Property   System.DateTime CreationTime {get;set;}
CreationTimeUtc   Property   System.DateTime CreationTimeUtc {get;set;}
Directory         Property   System.IO.DirectoryInfo Directory {get;}
DirectoryName     Property   System.String DirectoryName {get;}
```

```
Exists            Property   System.Boolean Exists {get;}
Extension         Property   System.String Extension {get;}
FullName          Property   System.String FullName {get;}
IsReadOnly        Property   System.Boolean IsReadOnly {get;set;}
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;}
Length            Property   System.Int64 Length {get;}
Name              Property   System.String Name {get;}
```

10. Use the membertype argument of the *Get-Member* cmdlet to view the methods available from the object returned by the *Get-ChildItem* cmdlet. To do this, supply a value of method to the membertype argument, as follows:

```
get-childitem | get-member -membertype method
```

11. The output from the previous list returns all the methods defined for the *Get-ChildItem* cmdlet. This output is shown here:

```
   TypeName: System.IO.DirectoryInfo

Name                      MemberType Definition
----                      ---------- ----------
Create                    Method     System.Void Create(), System.Void Creat...
CreateObjRef              Method     System.Runtime.Remoting.ObjRef CreateOb...
CreateSubdirectory        Method     System.IO.DirectoryInfo CreateSubdirect...
Delete                    Method     System.Void Delete(), System.Void Delet...
Equals                    Method     System.Boolean Equals(Object obj)
GetAccessControl          Method     System.Security.AccessControl.Directory...
GetDirectories            Method     System.IO.DirectoryInfo[] GetDirectorie...
GetFiles                  Method     System.IO.FileInfo[] GetFiles(String se...
GetFileSystemInfos        Method     System.IO.FileSystemInfo[] GetFileSyste...
GetHashCode               Method     System.Int32 GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetObjectData             Method     System.Void GetObjectData(Serialization...
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(DirectoryS...
```

```
set_Attributes            Method     System.Void set_Attributes(FileAttribut...
set_CreationTime          Method     System.Void set_CreationTime(DateTime v...
set_CreationTimeUtc       Method     System.Void set_CreationTimeUtc(DateTim...
set_LastAccessTime        Method     System.Void set_LastAccessTime(DateTime...
set_LastAccessTimeUtc     Method     System.Void set_LastAccessTimeUtc(DateT...
set_LastWriteTime         Method     System.Void set_LastWriteTime(DateTime ...
set_LastWriteTimeUtc      Method     System.Void set_LastWriteTimeUtc(DateTi...
ToString                  Method     System.String ToString()
```

12. Use the up arrow key to retrieve the previous *Get-ChildItem | Get-Member -MemberType* method command, and change the value method to *m** to use a wild card to retrieve the methods. The output will be exactly the same as the previous listing of members because the only membertype beginning with the letter m on the *Get-ChildItem* cmdlet is the *MemberType* method. The command is as follows:

```
get-childitem | get-member -membertype m*
```

13. Use the -inputobject argument to the *Get-Member* cmdlet to retrieve member definitions of each property or method in the list. The command to do this is as follows:

```
get-member -inputobject get-childitem
```

14. The output from the previous command is shown here:

```
PS C:\> get-member -inputobject get-childitem


   TypeName: System.String

Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone()
CompareTo        Method               System.Int32 CompareTo(Object value),...
Contains         Method               System.Boolean Contains(String value)
CopyTo           Method               System.Void CopyTo(Int32 sourceIndex,...
EndsWith         Method               System.Boolean EndsWith(String value)...
Equals           Method               System.Boolean Equals(Object obj), Sy...
GetEnumerator    Method               System.CharEnumerator GetEnumerator()
GetHashCode      Method               System.Int32 GetHashCode()
GetType          Method               System.Type GetType()
GetTypeCode      Method               System.TypeCode GetTypeCode()
get_Chars        Method               System.Char get_Chars(Int32 index)
get_Length       Method               System.Int32 get_Length()
IndexOf          Method               System.Int32 IndexOf(Char value, Int3...
IndexOfAny       Method               System.Int32 IndexOfAny(Char[] anyOf,...
Insert           Method               System.String Insert(Int32 startIndex...
IsNormalized     Method               System.Boolean IsNormalized(), System...
LastIndexOf      Method               System.Int32 LastIndexOf(Char value, ...
LastIndexOfAny   Method               System.Int32 LastIndexOfAny(Char[] an...
Normalize        Method               System.String Normalize(), System.Str...
PadLeft          Method               System.String PadLeft(Int32 totalWidt...
PadRight         Method               System.String PadRight(Int32 totalWid...
Remove           Method               System.String Remove(Int32 startIndex...
Replace          Method               System.String Replace(Char oldChar, C...
Split            Method               System.String[] Split(Params Char[] s...
StartsWith       Method               System.Boolean StartsWith(String valu...
```

```
Substring        Method                System.String Substring(Int32 startIn...
ToCharArray      Method                System.Char[] ToCharArray(), System.C...
ToLower          Method                System.String ToLower(), System.Strin...
ToLowerInvariant Method                System.String ToLowerInvariant()
ToString         Method                System.String ToString(), System.Stri...
ToUpper          Method                System.String ToUpper(), System.Strin...
ToUpperInvariant Method                System.String ToUpperInvariant()
Trim             Method                System.String Trim(Params Char[] trim...
TrimEnd          Method                System.String TrimEnd(Params Char[] t...
TrimStart        Method                System.String TrimStart(Params Char[]...
Chars            ParameterizedProperty System.Char Chars(Int32 index) {get;}
Length           Property              System.Int32 Length {get;}
```

15. This concludes the procedure for using the *Get-Member* cmdlet.

Quick Check

Q. To retrieve a listing of aliases beginning with the letter g that is sorted on the definition property, what command would you use?

A. *gal g* | sort-object -property definition*

Using the *New-Object* Cmdlet

The use of objects in Windows PowerShell provides many exciting opportunities to do things that are not "built into" the PowerShell. You may recall from using VBScript that there is an object called the wshShell object. If you are not familiar with this object, a drawing of the object model is shown in Figure 2-1.

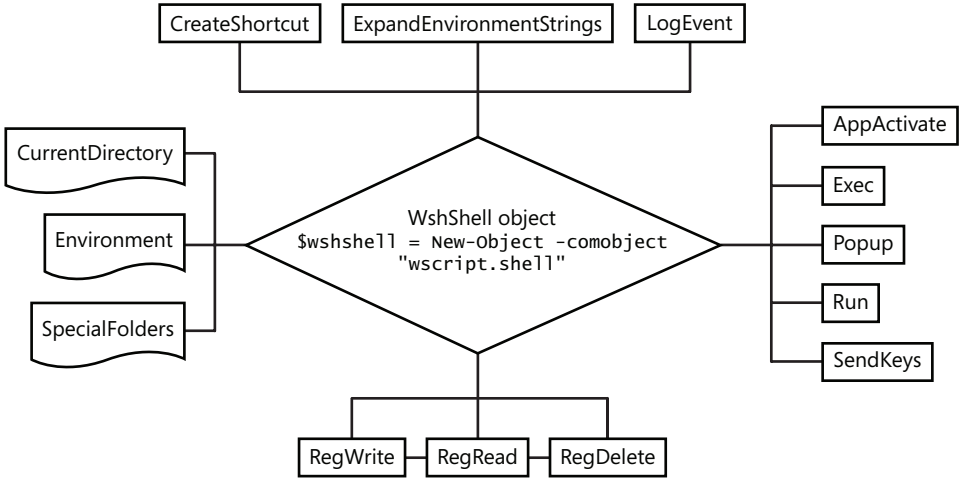

Figure 2-1 The VBScript wshShell object contributes many easy-to-use methods and properties for the network administrator

Just the Steps To create a new instance of the wshShell object, use the *New-Object* cmdlet while specifying the -comobject argument and supplying the program ID of "wscript.shell". Hold the object created in a variable. Example:

```
$wshShell = new-object -comobject "wscript.shell":
```

After the object has been created and stored in a variable, you can directly use any of the methods that are provided by the object. This is shown in the two lines of code that follow:

```
$wshShell = new-object -comobject "wscript.shell"
$wshShell.run("calc.exe")
```

In the previous code, we use the *New-Object* cmdlet to create an instance of the wshShell object. We then use the run method to launch Calculator. After the object is created and stored in the variable, you can use *Tab Completion* to suggest the names of the methods contained in the object. This is shown in Figure 2-2.

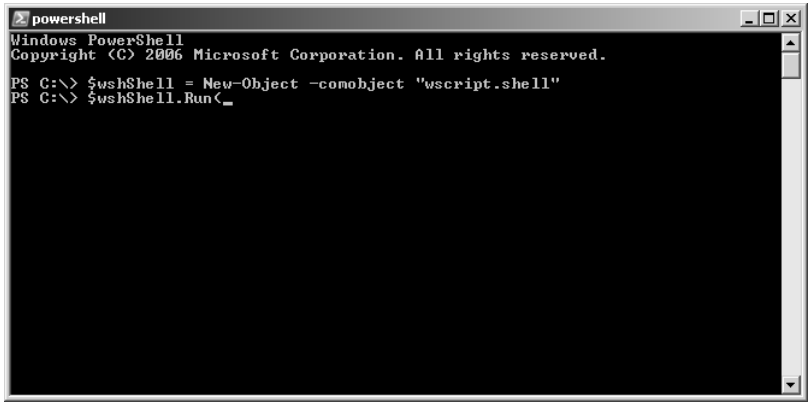

Figure 2-2 Tab Completion enumerates methods provided by the object

Creating the wshShell object

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. Create an instance of the wshShell object by using the *New-Object* cmdlet. Supply the comobject argument to the cmdlet, and specify the program ID for the wshShell object, which is "wscript.shell". Hold the object that is returned into a variable called *$wshShell*. The code to do this is as follows:

   ```
   $wshShell = new-object -comobject "wscript.shell"
   ```

3. Launch an instance of Calculator by using the run method from the wshShell object. Use *Tab Completion* to avoid having to type the entire name of the method. To use the method, begin the line with the variable you used to hold the wshShell object, followed by a period and the name of the method. Then supply the name of the program to run inside parentheses and quotes, as shown here:

```
$wshShell.run("Calc.exe")
```

4. Use the ExpandEnvironmentStrings method to print out the path to the Windows directory. It is stored in an environmental variable called *%windir%*. The *Tab Completion* feature of Windows PowerShell is useful for this method name. The environment variable must be contained in quotation marks, as shown here:

```
$wshShell.ExpandEnvironmentStrings("%windir%")
```

5. This command reveals the full path to the Windows directory on your machine. On my computer, the output looks like the following:

```
C:\WINDOWS
```

Creating a PowerShell Profile

As you create various aliases and functions, you may decide you like a particular key stroke combination and wish you could use your definition without always having to create it.

Tip I recommend reviewing the listing of all the aliases defined within Windows PowerShell before creating very many new aliases. The reason is that it will be easy, early on, to create duplicate settings (with slight variations).

Of course, you could create your own script that would perform your configuration if you remembered to run it; however, what if you wish to have a more standardized method of working with your profile? To do this, you need to create a custom profile that will hold your settings. The really useful feature of creating a Windows PowerShell profile is that after the profile is created, it loads automatically when PowerShell is launched. The steps for creating a Windows PowerShell profile are listed here:

Just the Steps Creating a Windows PowerShell profile

1. In a Windows PowerShell prompt, determine whether a profile exists by using the following command:

   ```
   test-path $profile
   ```

2. If tests-profile returns false, create a new profile file by using the following command:

   ```
   new-item -path $profile -itemtype file -force
   ```

3. Open the profile file in Notepad by using the following command:

   ```
   notepad $profile
   ```

4. Add the following toNotepad:

   A useful alias such as *gh* for *Get-Help*. This is shown here:

   ```
   Set-alias gh get-help
   ```

   A useful function to the profile such as one to open the profile in Notepad to allow for ease of editing the profile. This is shown here:

   ```
   function pro {notepad $profile}
   ```

5. When done editing, save the profile. Click Save As from the File menu, and ensure that you choose ALL Files in the dialog box to avoid saving the profile with a .txt extension. This is shown in Figure 2-3.

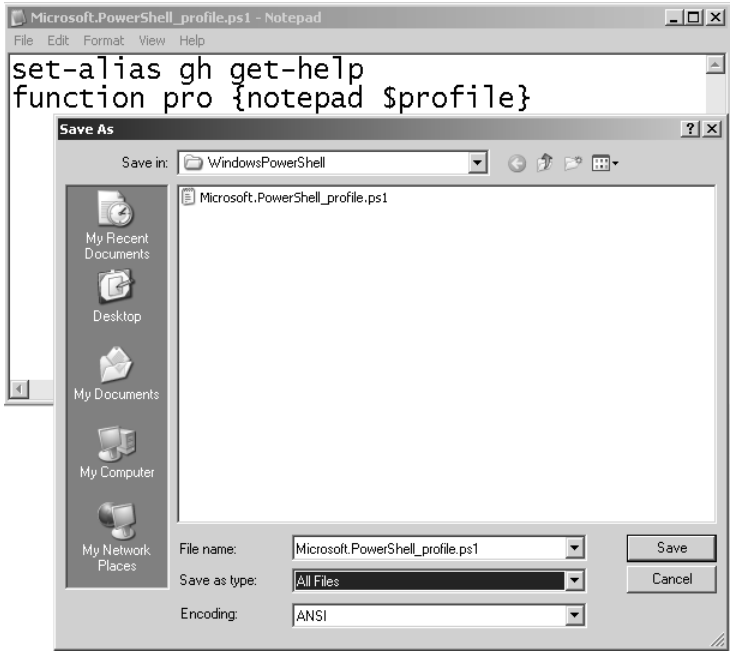

Figure 2-3 Ensure that Windows PowerShell can read the profile by saving it with the *All Files* option, under Save As Type, in Notepad

Just the Steps **Finding all aliases for a particular object** If you know the name of an object and you would like to retrieve all aliases for that object, you can use the *Get-Alias* cmdlet to retrieve the list of all aliases. Then you need to pipe the results to the *Where-Object* cmdlet and specify the value for the definition property. An example of doing this for the *Get-ChildItem* cmdlet is as follows:

```
gal | where-object {$_.definition -match "get-childitem"}
```

Working with Cmdlets: Step-by-Step Exercises

In this exercise, we explore the use of the *Get-ChildItem* and *Get-Member* cmdlets in Windows PowerShell. You will see that it is easy to use these cmdlets to automate routine administrative tasks. We also continue to experiment with the pipelining feature of Windows PowerShell.

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-Alias* cmdlet to retrieve a listing of all the aliases defined on the computer. Pipe this output to a *Where-Object* cmdlet. Specify a match argument against the definition property that matches the name of the *Get-ChildItem* cmdlet. The code is as follows:

   ```
   gal | where-object {$_.definition -match "get-childitem"}
   ```

3. The results from the previous command show three aliases defined for the *Get-ChildItem* cmdlet, as shown here:

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

4. Using the *gci* alias for the *Get-ChildItem* cmdlet, obtain a listing of files and folders contained in the root directory. This is shown here:

   ```
   gci
   ```

5. To identify large files more quickly, pipe the output to a *Where-Object* cmdlet, and specify the gt argument with a value of 1,000 to evaluate the length property. This is shown here:

   ```
   gci | where-object {$_.length -gt 1000}
   ```

6. To remove the cluttered data from your Windows PowerShell window, use *cls* to clear the screen. This is shown here:

   ```
   cls
   ```

7. Use the *Get-Alias* cmdlet to resolve the cmdlet to which the *cls* alias points. You can use the *gal* alias to avoid typing **get-alias** if you wish. This is shown here:

   ```
   gal cls
   ```

8. Use the *Get-Alias* cmdlet to resolve the cmdlet to which the *mred* alias points. This is shown here:

```
gal mred
```

9. It is likely that no *mred* alias is defined on your machine. In this case, you will see the following error message:

```
Get-Alias : Cannot find alias because alias 'mred' does not exist.
At line:1 char:4
+ gal <<<< mred
```

10. Use the *Clear-Host* cmdlet to clear the screen. This is shown here:

```
clear-host
```

11. Use the *Get-Member* cmdlet to retrieve a list of properties and methods from the *Get-ChildItem* cmdlet. This is shown here:

```
get-childitem | get-member -membertype property
```

12. The output from the above command is shown here. Examine the output, and identify a property that could be used with a *Where-Object* cmdlet to find the date that files have been modified.

```
Name              MemberType Definition
----              ---------- ----------
Attributes        Property   System.IO.FileAttributes Attributes {get;set;}
CreationTime      Property   System.DateTime CreationTime {get;set;}
CreationTimeUtc   Property   System.DateTime CreationTimeUtc {get;set;}
Directory         Property   System.IO.DirectoryInfo Directory {get;}
DirectoryName     Property   System.String DirectoryName {get;}
Exists            Property   System.Boolean Exists {get;}
Extension         Property   System.String Extension {get;}
FullName          Property   System.String FullName {get;}
IsReadOnly        Property   System.Boolean IsReadOnly {get;set;}
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;}
Length            Property   System.Int64 Length {get;}
Name              Property   System.String Name {get;}
```

13. Use the *Where-Object* cmdlet and choose the LastWriteTime property. This is shown here:

```
get-childitem | where-object {$_.LastWriteTime}
```

14. Use the up arrow and bring the previous command back up onto the command line. Now specify the gt argument and choose a recent date from your previous list of files, so you can ensure the query will return a result. My command looks like the following:

```
get-childitem | where-object {$_.LastWriteTime -gt "12/25/2006"}
```

15. Use the up arrow and retrieve the last command. Now direct the *Get-ChildItem* cmdlet to a specific folder on your hard drive, such as C:\fso, which may have been created in the

step-by-step exercise from Chapter 1. You can, of course, use any folder that exists on your machine. This command will look like the following:

```
get-childitem "C:\fso"| where-object {$_.LastWriteTime -gt "12/25/2006"}
```

16. Once again, use the up arrow and retrieve the last command. Add the recurse argument to the *Get-ChildItem* cmdlet. If your previous folder was not nested, then you may want to change to a different folder. You can, of course, use your Windows folder, which is rather deeply nested. I used my VBScript workshop folder, and the command is shown here (keep in mind that this command has wrapped and should be interpreted as a single line):

```
get-childitem -recurse "d:\vbsworkshop"| where-object
{$_.LastWriteTime -gt "12/25/2006" }
```

17. This concludes this step-by-step exercise. Completed commands for this exercise are in the StepByStep.txt file.

One Step Further: Working with *New-Object*

In this exercise, we create a couple of objects.

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. Create an instance of the wshNetwork object by using the *New-Object* cmdlet. Use the comobject argument, and give it the program ID for the wshNetwork object, which is "wscript.network". Store the results in a variable called *$wshnetwork*. The code looks like the following:

```
$wshnetwork = new-object -comobject "wscript.network"
```

3. Use the EnumPrinterConnections method from the wshNetwork object to print out a list of printer connections that are defined on your local computer. To do this, use the wshNetwork object that is contained in the *$wshnetwork* variable. The command for this is as follows:

```
$wshnetwork.EnumPrinterConnections()
```

4. Use the EnumNetworkDrives method from the wshNetwork object to print out a list of network connections that are defined on your local computer. To do this, use the wshNetwork object that is contained in the *$wshnetwork* variable. The command for this is as follows:

```
$wshnetwork.EnumNetworkDrives()
```

5. Use the up arrow twice and retrieve the *$wshnetwork.EnumPrinterConnections()* command. Use the *$colPrinters* variable to hold the collection of printers that is returned by the command. The code looks as follows:

```
$colPrinters = $wshnetwork.EnumPrinterConnections()
```

6. Use the up arrow and retrieve the *$wshnetwork.EnumNetworkDrives()* command. Use the Home key to move the insertion point to the beginning of the line. Modify the command so that it holds the collection of drives returned by the command into a variable called *$colDrives*. This is shown here:

```
$colDrives = $wshnetwork.EnumNetworkDrives()
```

7. Use the *$userName* variable to hold the name that is returned by querying the username property from the wshNetwork object. This is shown here:

```
$userName = $wshnetwork.UserName
```

8. Use the *$userDomain* variable to hold the name that is returned by querying the User-Domain property from the wshNetwork object. This is shown here:

```
$userDomain = $wshnetwork.UserDomain
```

9. Use the *$computerName* variable to hold the name that is returned by querying the User-Domain property from the wshNetwork object. This is shown here:

```
$computerName = $wshnetwork.ComputerName
```

10. Create an instance of the wshShell object by using the *New-Object* cmdlet. Use the comobject argument and give it the program ID for the wshShell object, which is "wscript.shell". Store the results in a variable called *$wshShell*. The code for this follows:

```
$wshShell = new-object -comobject "wscript.shell"
```

11. Use the Popup method from the wshShell object to produce a popup box that displays the domain name, user name, and computer name. The code for this follows:

```
$wshShell.Popup($userDomain+"\$userName $computerName")
```

12. Use the Popup method from the wshShell object to produce a popup box that displays the collection of printers held in the *$colPrinters* variable. The code looks as follows:

```
$wshShell.Popup($colPrinters)
```

13. Use the Popup method from the wshShell object to produce a popup box that displays the collection of drives held in the *$colDrives* variable. The code is as follows:

```
$wshShell.Popup($colDrives)
```

14. This concludes this one step further exercise. Completed commands for this exercise are in the OneStepFurther.txt file.

Chapter 2 Quick Reference

To	Do This
Produce a list of all the files in a folder	Use the *Get-ChildItem* cmdlet and supply a value for the folder
Produce a list of all the files in a folder and in the sub-folders	Use the *Get-ChildItem* cmdlet, supply a value for the folder, and specify the recurse argument
Produce a wide output of the results of a previous cmdlet	Use the appropriate cmdlet and pipe the resulting object to the *Format-Wide* cmdlet
Produce a listing of all the methods available from the *Get-ChildItem* cmdlet	Use the cmdlet and pipe the results into the *Get-Member* cmdlet. Use the -membertype argument and supply the Noun method
Produce a popup box	Create an instance of the wshShell object by using the *New-Object* cmdlet. Use the Popup method
Retrieve the currently logged-on user name	Create an instance of the wshNetwork object by using the *New-Object* cmdlet. Query the username property
Retrieve a listing of all currently mapped drives	Create an instance of the wshNetwork object by using the *New-Object* cmdlet. Use the EnumNetworkDrives method

Chapter 3

Leveraging PowerShell Providers

After completing this chapter, you will be able to:

- Understand the role of providers in Windows PowerShell
- Use the *Get-PSProvider* cmdlet
- Use the *Get-PSDrive* cmdlet
- Use the *Get-Item* cmdlet
- Use the *Set-Location* cmdlet
- Use the file system model to access data from each of the built-in providers

Windows PowerShell provides a consistent way to access information external to the shell environment. To do this, it uses providers. These providers are actually .NET programs that hide all the ugly details to provide an easy way to access information. The beautiful thing about the way the provider model works is that all the different sources of information are accessed in exactly the same manner. This chapter demonstrates how to leverage the PowerShell providers. All the scripts mentioned in this chapter can be found in the corresponding scripts folder on the CD.

Identifying the Providers

By identifying the providers installed with Windows PowerShell, we can begin to understand the capabilities intrinsic to a default installation. Providers expose information contained in different data stores by using a drive and file system analogy. An example of this is obtaining a listing of registry keys—to do this, you would connect to the registry "drive" and use the *Get-ChildItem* cmdlet, which is exactly the same method you would use to obtain a listing of files on the hard drive. The only difference is the specific name associated with each drive. Providers can be created by anyone familiar with Windows .NET programming. When a new provider is created, it is called a *snap-in*. A snap-in is a dynamic link library (*dll*) file that must be installed into Windows PowerShell. After a snap-in has been installed, it cannot be un-installed—however, the snap-in can be removed from the current Windows PowerShell console.

Just the Steps To obtain a listing of all the providers, use the *Get-PSProvider* cmdlet. Example: get-psprovider. This command produces the following list on a default installation of the Windows PowerShell:

```
Name                 Capabilities                 Drives
----                 ------------                 ------
Alias                ShouldProcess                {Alias}
Environment          ShouldProcess                {Env}
FileSystem           Filter, ShouldProcess        {C, D, E, F...}
Function             ShouldProcess                {Function}
Registry             ShouldProcess                {HKLM, HKCU}
Variable             ShouldProcess                {Variable}
Certificate          ShouldProcess                {cert}
```

Understanding the Alias Provider

In Chapter 1, Overview of Windows PowerShell, we presented the various *Help* utilities available that show how to use cmdlets. The alias provider provides easy-to-use access to all aliases defined in Windows PowerShell. To work with the aliases on your machine, use the *Set-Location* cmdlet and specify the Alias:\ drive. You can then use the same cmdlets you would use to work with the file system.

Tip With the alias provider, you can use a *Where-Object* cmdlet and filter to search for an alias by name or description.

Working with the alias provider

1. Open Windows PowerShell.
2. Obtain a listing of all the providers by using the *Get-PSProvider* cmdlet. This is shown here:

   ```
   Get-PSProvider
   ```

3. The PSDrive associated with the alias provider is called Alias. This is seen in the listing produced by the *Get-PSProvider* cmdlet. Use the *Set-Location* cmdlet to change to the Alias drive. Use the *sl* alias to reduce typing. This command is shown here:

   ```
   sl alias:\
   ```

4. Use the *Get-ChildItem* cmdlet to produce a listing of all the aliases that are defined on the system. To reduce typing, use the alias *gci* in place of *Get-ChildItem*. This is shown here:

   ```
   GCI
   ```

5. Use a *Where-Object* cmdlet filter to reduce the amount of information that is returned by using the *Get-ChildItem* cmdlet. Produce a listing of all the aliases that begin with the letter s. This is shown here:

   ```
   GCI | Where-Object {$_.name -like "s*"}
   ```

6. To identify other properties that could be used in the filter, pipeline the results of the *Get-ChildItem* cmdlet into the *Get-Member* cmdlet. This is shown here:

```
Get-ChildItem |Get-Member
```

7. Press the up arrow twice, and edit the previous filter to include only definitions that contain the word set. The modified filter is shown here:

```
GCI | Where-Object {$_.definition -like "set*"}
```

8. The results of this command are shown here:

```
CommandType     Name                           Definition
-----------     ----                           ----------
Alias           sal                            Set-Alias
Alias           sc                             Set-Content
Alias           si                             Set-Item
Alias           sl                             Set-Location
Alias           sp                             Set-ItemProperty
Alias           sv                             Set-Variable
Alias           cd                             Set-Location
Alias           chdir                          Set-Location
Alias           set                            Set-Variable
```

9. Press the up arrow three times, and edit the previous filter to include only names of aliases that are like the letter w. This revised command is seen here:

```
GCI | Where-Object {$_.name -like "*w*"}
```

10. The results from this command are similar to those shown here:

```
CommandType     Name                           Definition
-----------     ----                           ----------
Alias           fw                             Format-Wide
Alias           gwmi                           Get-WmiObject
Alias           where                          Where-Object
Alias           write                          Write-Output
Alias           pwd                            Get-Location
```

11. From the list above, note that *where* is an alias for the *Where-Object* cmdlet. Press the up arrow one time to retrieve the previous command. Edit it to use the *where* alias instead of spelling out the entire *Where-Object* cmdlet name. This revised command is seen here:

```
GCI | where {$_.name -like "*w*"}
```

Caution When using the *Set-Location* cmdlet to switch to a newly created PSDrive, you must follow the name of the PSDrive with a colon. A trailing forward slash or backward slash is optional. An error will be generated if the colon is left out, as shown in Figure 3-1. I prefer to use the backward slash (\) because it is consistent with normal Windows file system operations.

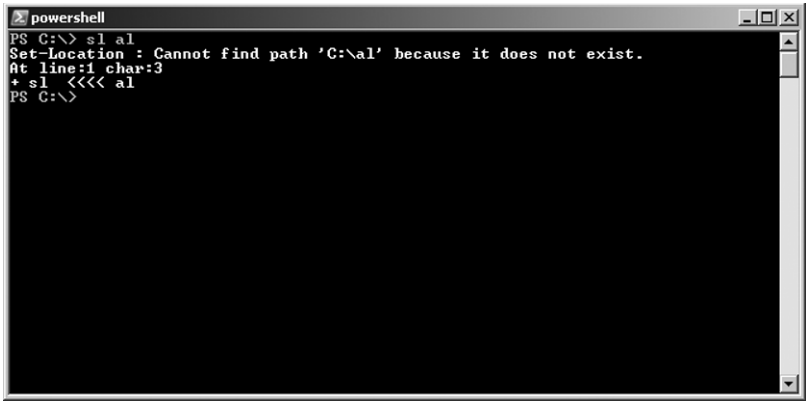

Figure 3-1 Using Set-Location without : results in an error

Understanding the Certificate Provider

In the preceding section, we explored working with the alias provider. Because the file system model applies to the certificate provider in much the same way as it did the alias provider, many of the same cmdlets can be used. To find information about the certificate provider, use the *Get-Help* cmdlet. If you are unsure what articles in *Help* may be related to certificates, you can use the wild card asterisk (*) parameter. This command is shown here:

```
get-help *cer*
```

The certificate provider gives you the ability to sign scripts and allows Windows PowerShell to work with signed and unsigned scripts as well. It also gives you the ability search for, copy, move, and delete certificates. Using the certificate provider, you can even open the Certificates Microsoft Management Console (MMC). The commands used in the procedure are in the ObtainingAListingOfCertificates.txt file.

Obtaining a listing of certificates

1. Open Windows PowerShell.
2. Set your location to the cert PSDrive. To do this, use the *Set-Location* cmdlet, as shown here:

   ```
   Set-Location cert:\
   ```

3. Use the *Get-ChildItem* cmdlet to produce a list of the certificates, as shown here:

   ```
   Get-ChildItem
   ```

4. The list produced is shown here:

   ```
   Location   : CurrentUser
   StoreNames : {?, UserDS, AuthRoot, CA...}

   Location   : LocalMachine
   StoreNames : {?, AuthRoot, CA, AddressBook...}
   ```

5. Use the -recurse argument to cause the *Get-ChildItem* cmdlet to produce a list of all the certificate stores. To do this, press the up arrow key one time, and add the -recurse argument to the previous command. This is shown here:

```
Get-ChildItem -recurse
```

6. Use the -path argument for *Get-ChildItem* to produce a listing of certificates in another store, without having to use the *Set-Location* cmdlet to change your current location. Using the *gci* alias, the command is shown here:

```
GCI -path currentUser
```

7. Your listing of certificate stores will look similar to the one shown here:

```
Name : ?

Name : UserDS

Name : AuthRoot

Name : CA

Name : AddressBook

Name : ?

Name : Trust

Name : Disallowed

Name : _NMSTR

Name : ?????k

Name : My

Name : Root

Name : TrustedPeople

Name : ACRS

Name : TrustedPublisher

Name : REQUEST
```

8. Change your working location to the currentuser\authroot certificate store. To do this, use the *sl* alias followed by the path to the certificate store. This command is shown here:

```
sl currentuser\authroot
```

9. Use the *Get-ChildItem* cmdlet to produce a listing of certificates in the currentuser\authroot certificate store that contain the name C&W in the subject field. Use the *gci* alias to reduce the amount of typing. Pipeline the resulting object to a *Where-Object* cmdlet, but use the *where* alias instead of typing *Where-Object*. The code to do this is shown here:

```
GCI | where {$_.subject -like "*c&w*"}
```

10. On my machine, there are four certificates listed. These are shown here:

```
Thumbprint                                Subject
----------                                -------
F88015D3F98479E1DA553D24FD42BA3F43886AEF  O=C&W HKT SecureNet CA SGC Root, C=hk
9BACF3B664EAC5A17BED08437C72E4ACDA12F7E7  O=C&W HKT SecureNet CA Class A, C=hk
4BA7B9DDD68788E12FF852E1A024204BF286A8F6  O=C&W HKT SecureNet CA Root, C=hk
47AFB915CDA26D82467B97FA42914468726138DD  O=C&W HKT SecureNet CA Class B, C=hk
```

11. Use the up arrow, and edit the previous command so that it will return only certificates that contain the phrase *SGC Root* in the subject property. The revised command is shown here:

```
GCI | where {$_.subject -like "*SGC Root*"}
```

12. The resulting output on my machine contains an additional certificate. This is shown here:

```
Thumbprint                                Subject
----------                                -------
F88015D3F98479E1DA553D24FD42BA3F43886AEF  O=C&W HKT SecureNet CA SGC Root, C=hk
687EC17E0602E3CD3F7DFBD7E28D57A0199A3F44  O=SecureNet CA SGC Root, C=au
```

13. Use the up arrow, and edit the previous command. This time, change the *Where-Object* cmdlet so that it filters on the thumbprint attribute that is equal to F88015D3F98479E1DA553D24FD42BA3F43886AEF. You do not have to type that, however; to copy the thumbprint, you can highlight it and press Enter in Windows PowerShell, as shown in Figure 3-2. The revised command is shown here:

```
GCI | where {$_.thumbprint -eq "F88015D3F98479E1DA553D24FD42BA3F43886AEF"}
```

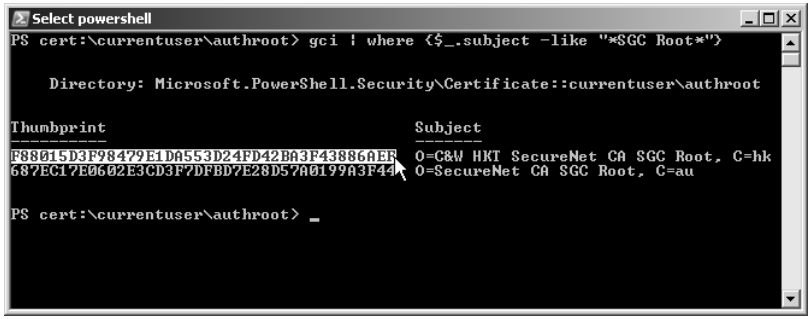

Figure 3-2 Highlight items to copy using the mouse

Troubleshooting If copying from inside a Windows PowerShell window does not work, then you probably need to enable Quick Edit Mode. To do this, right-click the PowerShell icon in the upper left-hand corner of the Windows PowerShell window. Choose Properties, and select Quick Edit Mode. This is shown in Figure 3-3.

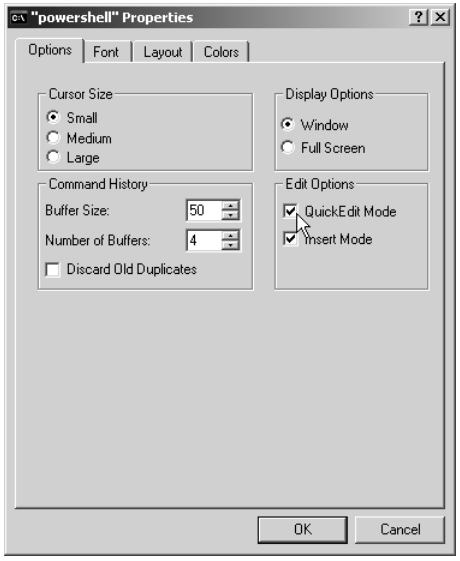

Figure 3-3 Enable Quick Edit Mode to enable Clipboard Support

14. To see all the properties of the certificate, pipeline the certificate object to a *Format-List* cmdlet and choose all the properties. The revised command is shown here:

```
GCI | where {$_.thumbprint -eq "F88015D3F98479E1DA553D24FD42BA3F43886AEF"} |
Format-List *
```

15. The output contains all the properties of the certificate object and is shown here:

```
PSPath            : Microsoft.PowerShell.Security\Certificate::currentuser\aut
                    hroot\F88015D3F98479E1DA553D24FD42BA3F43886AEF
PSParentPath      : Microsoft.PowerShell.Security\Certificate::currentuser\aut
                    hroot
PSChildName       : F88015D3F98479E1DA553D24FD42BA3F43886AEF
PSDrive           : cert
PSProvider        : Microsoft.PowerShell.Security\Certificate
PSIsContainer     : False
Archived          : False
Extensions        : {}
FriendlyName      : CW HKT SecureNet CA SGC Root
IssuerName        : System.Security.Cryptography.X509Certificates.X500Distingu
                    ishedName
NotAfter          : 10/16/2009 5:59:00 AM
NotBefore         : 6/30/1999 6:00:00 AM
HasPrivateKey     : False
PrivateKey        :
```

```
PublicKey          : System.Security.Cryptography.X509Certificates.PublicKey
RawData            : {48, 130, 2, 235...}
SerialNumber       : 00
SubjectName        : System.Security.Cryptography.X509Certificates.X500Distingu
                     ishedName
SignatureAlgorithm : System.Security.Cryptography.Oid
Thumbprint         : F88015D3F98479E1DA553D24FD42BA3F43886AEF
Version            : 1
Handle             : 75655840
Issuer             : O=C&W HKT SecureNet CA SGC Root, C=hk
Subject            : O=C&W HKT SecureNet CA SGC Root, C=hk
```

16. Open the Certificates MMC. This MMC is called Certmgr.msc and can be launched by simply typing the name inside Windows PowerShell, as shown here:

```
Certmgr.msc
```

17. But it is more fun to use the *Invoke-Item* cmdlet to launch the Certificates MMC. To do this, supply the PSDrive name of cert:\ to the *Invoke-Item* cmdlet. This is shown here:

```
Invoke-Item cert:\
```

18. Compare the information obtained from Windows PowerShell with the information displayed in the Certificates MMC. They are the same. The certificate is shown in Figure 3-4.

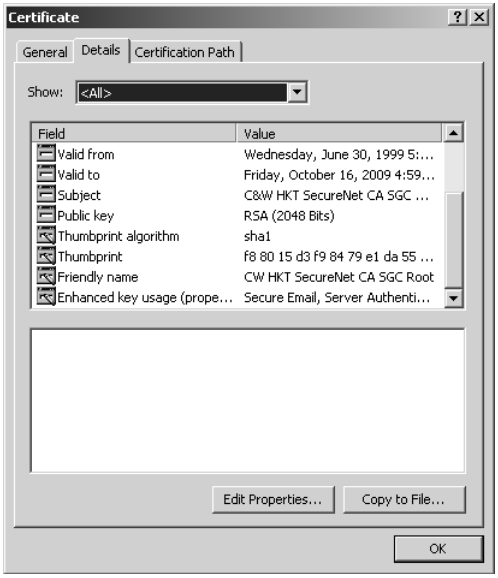

Figure 3-4 Certmgr.msc can be used to examine certificate properties

19. This concludes this procedure.

Understanding the Environment Provider

The environment provider in Windows PowerShell is used to provide access to the system environment variables. If you open a CMD (command) shell and type **set**, you will obtain a listing of all the environment variables defined on the system. If you use the echo command in the CMD shell to print out the value of %windir%, you will obtain the results seen in Figure 3-5.

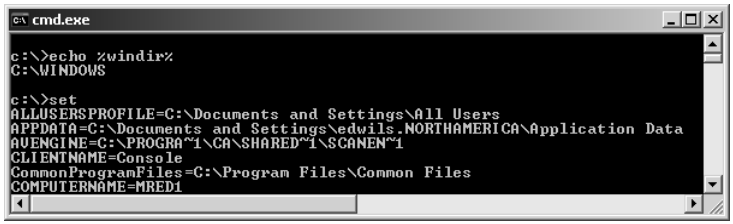

Figure 3-5 Use *set* in a CMD prompt to see environment variables

Environment variables are used by various applications and other utilities as a shortcut to provide easy access to specific files, folders, and configuration data. By using the environment provider in Windows PowerShell, you can obtain a listing of the environment variables. You can also add, change, clear, and delete these variables.

Obtaining a listing of environment variables

1. Open Windows PowerShell.
2. Obtain a listing of the PSDrives by using the *Get-PSDrive* cmdlet. This is shown here:

   ```
   Get-PSDrive
   ```

3. Note that the Environment PSDrive is called *env*. Use the *env* name with the *Set-Location* cmdlet and change to the environment PSDrive. This is shown here:

   ```
   Set-Location env:\
   ```

4. Use the *Get-Item* cmdlet to obtain a listing of all the environment variables on the system. This is shown here:

   ```
   Get-Item *
   ```

5. Use the *Sort-Object* cmdlet to produce an alphabetical listing of all the environment variables by name. Use the up arrow to retrieve the previous command, and pipeline the returned object into the *Sort-Object* cmdlet. Use the property argument, and supply name as the value. This command is shown here:

   ```
   get-item * | Sort-Object  -property name
   ```

6. Use the *Get-Item* cmdlet to retrieve the value associated with the environment variable *windir*. This is shown here:

```
get-item windir
```

7. Use the up arrow and retrieve the previous command. Pipeline the object returned to the *Format-List* cmdlet and use the wild card character to print out all the properties of the object. The modified command is shown here:

```
get-item windir | Format-List *
```

8. The properties and their associated values are shown here:

```
PSPath        : Microsoft.PowerShell.Core\Environment::windir
PSDrive       : Env
PSProvider    : Microsoft.PowerShell.Core\Environment
PSIsContainer : False
Name          : windir
Key           : windir
Value         : C:\WINDOWS
```

9. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Creating a new environment variable

1. You should still be in the Environment PSDrive from the previous procedure. If not, use the *Set-Location env:* command).

2. Use the *Get-Item* cmdlet to produce a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet using the property of name. To reduce typing, use the *gi* alias and the *sort* alias. This is shown here:

```
GI * | Sort -Property Name
```

3. Use the *New-Item* cmdlet to create a new environment variable. The path argument will be dot (.) because you are already on the env:\ PSDrive. The -name argument will be admin, and the value argument will be your given name. The completed command is shown here:

```
New-Item -Path . -Name admin -Value mred
```

4. Use the *Get-Item* cmdlet to ensure the *admin* environment variable was properly created. This command is shown here:

```
Get-Item admin
```

5. The results of the previous command are shown here:

```
Name                           Value
----                           -----
admin                          mred
```

6. Use the up arrow to retrieve the previous command. Pipeline the results to the *Format-List* cmdlet, and choose All Properties. This command is shown here:

```
Get-Item admin | Format-List *
```

7. The results of the previous command include the PSPath, PSDrive, and additional information about the newly created environment variable. These results are shown here:

```
PSPath        : Microsoft.PowerShell.Core\Environment::admin
PSDrive       : Env
PSProvider    : Microsoft.PowerShell.Core\Environment
PSIsContainer : False
Name          : admin
Key           : admin
Value         : mred
```

8. This concludes this procedure. Leave PowerShell open for the next procedure.

Renaming an environment variable

1. Use the *Get-ChildItem* cmdlet to obtain a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet and sort the list on the name property. Use the *gci* and *sort* aliases to reduce typing. The code to do this is shown here:

```
GCI | Sort -Property name
```

2. The *admin* environment variable should be near the top of the list of system variables. If it is not, then create it by using the *New-Item* cmdlet. The path argument has a value of dot (.); the name argument has the value of admin; and the value argument should be the user's given name. If this environment variable was created in the previous exercise, then PowerShell will report that it already exists. This is shown here:

```
New-Item -Path . -Name admin -Value mred
```

3. Use the *Rename-Item* cmdlet to rename the *admin* environment variable to *super*. The path argument combines both the PSDrive name and the environment variable name. The NewName argument is the desired new name without the PSDrive specification. This command is shown here:

```
Rename-Item -Path env:admin -NewName super
```

4. To verify that the old environment variable *admin* has been renamed *super*, press the up arrow two or three times to retrieve the *gci | sort -property name* command. This is command is shown here:

```
GCI | Sort -Property name
```

5. This concludes this procedure. Do not close the Windows PowerShell. Leave it open for the next procedure.

Removing an environment variable

1. Use the *Get-ChildItem* cmdlet to obtain a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet and sort the list on the name property. Use the *gci* and *sort* aliases to reduce typing. The code to do this is shown here:

```
GCI | Sort -Property name
```

2. The *super* environment variable should be in the list of system variables. If it is not, then create it by using the *New-Item* cmdlet. The path argument has a value of dot (.); the name argument has the value of super; and the value argument should be the user's given name. If this environment variable was created in the previous exercise, then PowerShell will report that it already exists. This is shown here:

```
New-Item -Path . -Name super -Value mred
```

3. Use the *Remove-Item* cmdlet to remove the *super* environment variable. The name of the item to be removed is typed following the name of the cmdlet. If you are still in the env:\ PSDrive, you will not need to supply a -path argument. The command is shown here:

```
Remove-Item super
```

4. Use the *Get-ChildItem* cmdlet to verify that the environment variable *super* has been removed. To do this, press the up arrow 2 or 3 times to retrieve the *gci | sort -property name* command. This command is shown here:

```
GCI | Sort -Property name
```

5. This concludes this procedure.

Understanding File System Provider

The file system provider is the easiest Windows PowerShell provider to understand—it provides access to the file system. When Windows PowerShell is launched, it automatically opens on the C:\PSDrive. Using the Windows PowerShell filesystem provider, you can create both directories and files. You can retrieve properties of files and directories, and you can delete them as well. In addition, you can open files and append or overwrite data to the files. This can be done with inline code, or by using the pipelining feature of Windows PowerShell. The commands used in the procedure are in the IdentifyingPropertiesOfDirectories.txt, CreatingFoldersAndFiles.txt, and ReadingAndWritingForFiles.txt files.

Working with directory listings

1. Open Windows PowerShell.
2. Use the *Get-ChildItem* cmdlet to obtain a directory listing of the C:\ drive. Use the *gci* alias to reduce typing. This is shown here:

```
GCI C:\
```

3. Use the up arrow to retrieve the *gci C:* command. Pipeline the object created into a *Where-Object* cmdlet, and look for containers. This will reduce the output to only directories. The modified command is shown here:

```
GCI C:\ | where {$_.psiscontainer}
```

4. Use the up arrow to retrieve the *gci C:\ | where {$_.psiscontainer}* command and use the exclamation point (!), meaning *not*, to retrieve only items in the PSDrive that are not directories. The modified command is shown here:

```
GCI C:\ | where {!$_.psiscontainer}
```

5. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Identifying properties of directories

1. Use the *Get-ChildItem* cmdlet and supply a value of *C:* for the path argument. Pipeline the resulting object into the *Get-Member* cmdlet. Use the *gci* and *gm* aliases to reduce typing. This command is shown here:

```
GCI  -Path C:\ | GM
```

2. The resulting output contains methods, properties, and more. Filter the output by pipelining the output into a *Where-Object* cmdlet and specifying the *membertype* attribute as equal to property. To do this, use the up arrow to retrieve the previous *gci -path C:\ | gm* command. Pipeline the resulting object into the *Where-Object* cmdlet and filter on the *membertype* attribute. The resulting command is shown here:

```
GCI  -Path C:\ | GM | Where {$_.membertype -eq "property"}
```

3. The previous *gci -path C:\ | gm | where {$_.membertype -eq "property"}* command returns information on both the System.IO.DirectoryInfo and the System.IO.FileInfo objects. To reduce the output to only the properties associated with the System.IO.FileInfo object, we need to use a compound *Where-Object* cmdlet. Use the up arrow to retrieve the *gci -path C:\ | gm | where {$_.membertype -eq "property"}* command. Add the And conjunction and retrieve objects that have a typename that is like *file*. The modified command is shown here:

```
GCI  -Path C:\ | GM | where {$_.membertype -eq "property" -AND $_.typename -like
"*file*"}
```

4. The resulting output only contains the properties for a System.IO.FileInfo object. These properties are shown here:

```
   TypeName: System.IO.FileInfo

Name              MemberType Definition
----              ---------- ----------
Attributes        Property   System.IO.FileAttributes Attributes {get;set;}
CreationTime      Property   System.DateTime CreationTime {get;set;}
CreationTimeUtc   Property   System.DateTime CreationTimeUtc {get;set;}
```

```
Directory         Property   System.IO.DirectoryInfo Directory {get;}
DirectoryName     Property   System.String DirectoryName {get;}
Exists            Property   System.Boolean Exists {get;}
Extension         Property   System.String Extension {get;}
FullName          Property   System.String FullName {get;}
IsReadOnly        Property   System.Boolean IsReadOnly {get;set;}
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;}
Length            Property   System.Int64 Length {get;}
Name              Property   System.String Name {get;}
```

5. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Creating folders and files

1. Use the *Get-Item* cmdlet to obtain a listing of files and folders. Pipeline the resulting object into the *Where-Object* cmdlet and use the PsIsContainer property to look for folders. Use the name property to find names that contain the word *my* in them. Use the *gi* alias and the *where* alias to reduce typing. The command is shown here:

```
GI * | Where {$_.PsisContainer -AND $_.name -Like "*my*"}
```

2. If you were following along in the previous chapters, you will have a folder called Mytest off the root of the C:\ drive. Use the *Remove-Item* cmdlet to remove the Mytest folder. Specify the recurse argument to also delete files contained in the C:\Mytest folder. If your location is still set to Env, then change it to C or search for C:\Mytest. The command is shown here:

```
RI mytest -recurse
```

3. Press the up arrow twice and retrieve the *gi * | where {$_.PsisContainer -AND $_.name -Like "*my*"}* command to confirm the folder was actually deleted. This command is shown here:

```
GI * | Where {$_.PsisContainer -AND $_.name -Like "*my*"}
```

4. Use the *New-Item* cmdlet to create a folder named Mytest. Use the path argument to specify the path of C:\. Use the name argument to specify the name of Mytest, and use the type argument to tell Windows PowerShell the new item will be a directory. This command is shown here:

```
New-Item -Path C:\ -Name mytest -Type directory
```

5. The resulting output, shown here, confirms the operation:

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

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----          1/4/2007   2:43 AM            mytest
```

6. Use the *New-Item* cmdlet to create an empty text file. To do this, use the up arrow and retrieve the previous *new-item -path C:\ -name Mytest -type directory* command. Edit the path argument so that it is pointing to the C:\Mytest directory. Edit the name argument to specify a text file named Myfile, and specify the type argument as file. The resulting command is shown here:

```
New-Item -Path C:\mytest -Name myfile.txt -type file
```

7. The resulting message, shown here, confirms the creation of the file:

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

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          1/4/2007   3:12 AM          0 myfile.txt
```

8. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Reading and writing for files

1. Delete Myfile.txt (created in the previous procedure). To do this, use the *Remove-Item* cmdlet and specify the path argument as C:\Mytest\Myfile.txt. This command is shown here:

```
RI -Path C:\mytest\myfile.txt
```

2. Use the up arrow twice to retrieve the *new-item -path C:\Mytest -name Myfile.txt -type* file. Add the -value argument to the end of the command line, and supply a value of *my file*. This command is shown here:

```
New-Item -Path C:\mytest -Name myfile.txt -Type file -Value "My file"
```

3. Use the *Get-Content* cmdlet to read the contents of Myfile.txt. This command is shown here:

```
Get-Content C:\mytest\myfile.txt
```

4. Use the *Add-Content* cmdlet to add additional information to the Myfile.txt file. This command is shown here:

```
Add-Content C:\mytest\myfile.txt -Value "ADDITIONAL INFORMATION"
```

5. Press the up arrow twice and retrieve the *get-content C:\Mytest\Myfile.txt* command, which is shown here:

```
Get-Content C:\mytest\myfile.txt
```

6. The output from the *get-content C:\Mytest\Myfile.txt* command is shown here:

```
My fileADDITIONAL INFORMATION
```

7. Press the up arrow twice, and retrieve the *add-content C:\mytest\Myfile.txt -value* "ADDITIONAL INFORMATION" command to add additional information to the file. This command is shown here:

```
Add-Content C:\mytest\myfile.txt -Value "ADDITIONAL INFORMATION"
```

8. Use the up arrow to retrieve the *get-content C:\Mytest\Myfile.txt* command, which is shown here:

```
Get-Content C:\mytest\myfile.txt
```

9. The output produced is shown here. Notice that the second time, the "*ADDITIONAL INFORMATION*" command was added to a new line.

```
My fileADDITIONAL INFORMATION
ADDITIONAL INFORMATION
```

10. Use the *Set-Information* cmdlet to overwrite the contents of the Myfile.txt file. Specify the value argument as "*Setting information*". This command is shown here:

```
Set-Content C:\mytest\myfile.txt -Value "Setting information"
```

11. Use the up arrow to retrieve the *get-content C:\Mytest\Myfile.txt* command, which is shown here:

```
Get-Content C:\mytest\myfile.txt
```

12. The output from the *Get-Content* command is shown here:

```
Setting information
```

13. This concludes this procedure.

Understanding the Function Provider

The Function provider provides access to the functions defined in Windows PowerShell. By using the function provider you can obtain a listing of all the functions on your system. You can also add, modify, and delete functions. The function provider uses a file system–based model, and the cmdlets learned earlier also apply to working with functions. The commands used in the procedure are in the ListingAllFunctionsOnTheSystem.txt file.

Listing all functions on the system

1. Open Windows PowerShell.
2. Use the *Set-Location* cmdlet to change the working location to the function PSDrive. This command is shown here:

```
Set-Location function:\
```

3. Use the *Get-ChildItem* cmdlet to enumerate all the functions. Do this by using the *gci* alias, as shown here:

```
GCI
```

4. The resulting list contains many functions that use *Set-Location* to the different drive letters. A partial view of this output is shown here:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Function        prompt                          'PS ' + $(Get-Location) + $(...
Function        TabExpansion                    ...
Function        Clear-Host                      $spaceType = [System.Managem...
Function        more                            param([string[]]$paths);  if...
Function        help                            param([string]$Name,[string[...
Function        man                             param([string]$Name,[string[...
Function        mkdir                           param([string[]]$paths); New...
Function        md                              param([string[]]$paths); New...
Function        A:                              Set-Location A:
Function        B:                              Set-Location B:
Function        C:                              Set-Location C:
Function        D:                              Set-Location D:
```

5. To return only the functions that are used for drives, use the *Get-ChildItem* cmdlet and pipe the object returned into a *Where-Object* cmdlet. Use the default *$_* variable to filter on the definition attribute. Use the like argument to search for definitions that contain the word *set*. The resulting command is shown here:

```
GCI | Where {$_.definition -like "set*"}
```

6. If you are more interested in functions that are not related to drive mappings, then you can use the notlike argument instead of like. The easiest way to make this change is to use the up arrow and retrieve the *gci | where {$_.definition -like "set*"}* and then change the filter from *like* to *notlike*. The resulting command is shown here:

```
GCI | Where {$_.definition -notlike "set*"}
```

7. The resulting listing of functions is shown here:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Function        prompt                          'PS' + $(Get-Location) + $(...
Function        TabExpansion                    ...
Function        Clear-Host                      $spaceType = [System.Managem...
Function        more                            param([string[]]$paths);  if...
Function        help                            param([string]$Name,[string[...
Function        man                             param([string]$Name,[string[...
Function        mkdir                           param([string[]]$paths); New...
Function        md                              param([string[]]$paths); New...
Function        pro                             notepad $profile
```

8. Use the *Get-Content* cmdlet to retrieve the text of the *md* function. This is shown here:

```
Get-Content md
```

9. The content of the *md* function is shown here:

```
param([string[]]$paths); New-Item -type directory -path $paths
```

10. This concludes this procedure.

Understanding the Registry Provider

The registry provider provides a consistent and easy way to work with the registry from within Windows PowerShell. Using the registry provider, you can search the registry, create new registry keys, delete existing registry keys, and modify values and access control lists (ACLs) from within Windows PowerShell. The commands used in the procedure are in the UnderstandingTheRegistryProvider.txt file. Two PSDrives are created by default. To identify the PSDrives that are supplied by the registry provider, you can use the *Get-PSDrive* cmdlet, pipeline the resulting objects into the *Where-Object* cmdlet, and filter on the provider property while supplying a value that is like the word registry. This command is shown here:

```
get-psDrive | where {$_.Provider -like "*Registry*"}
```

The resulting list of PSDrives is shown here:

```
Name       Provider      Root                                    CurrentLocation
----       --------      ----                                    ---------------
HKCU       Registry      HKEY_CURRENT_USER
HKLM       Registry      HKEY_LOCAL_MACHINE
```

Obtaining a listing of registry keys

1. Open Windows PowerShell.
2. Use the *Get-ChildItem* cmdlet and supply the HKLM:\ PSDrive as the value for the path argument. Specify the software key to retrieve a listing of software applications on the local machine. The resulting command is shown here:

   ```
   GCI -path HKLM:\software
   ```

3. A partial listing of similar output is shown here. The corresponding keys, as seen in Regedit.exe, are shown in Figure 3-6.

   ```
      Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software

   SKC  VC Name                           Property
   ---  -- ----                           --------
     2   0 781                            {}
     1   0 8ec                            {}
     4   0 Adobe                          {}
    12   0 Ahead                          {}
     2   1 Analog Devices                 {ProductDir}
     2   0 Andrea Electronics             {}
     1   0 Application Techniques         {}
   ```

4. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

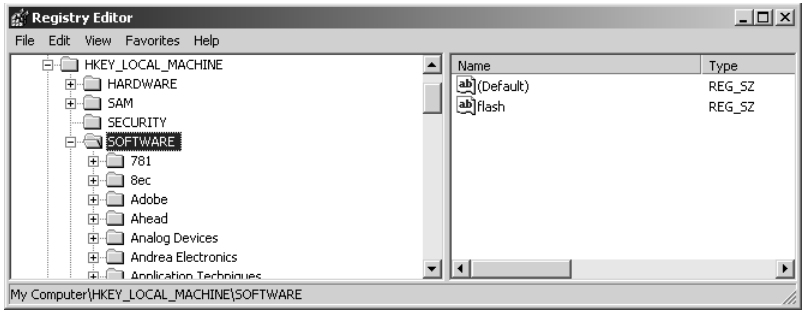

Figure 3-6 A Regedit.exe similar view of HKEY_LOCAL_MACHINE\SOFTWARE

Searching for hotfixes

1. Use the *Get-ChildItem* cmdlet and supply a value for the path argument. Use the HKLM:\ PSDrive and supply a path of Software\Microsoft\Windows NT\CurrentVersion\Hotfix. Because there is a space in Windows NT, you will need to use a single quote (') to encase the command. You can use *Tab completion* to assist with the typing. The completed command is shown here:

```
GCI -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\HotFix'
```

2. The resulting similar list of hotfixes is seen in the output here, in abbreviated fashion:

```
   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Micros
oft\Windows NT\CurrentVersion\HotFix

SKC  VC Name                           Property
---  -- ----                           --------
  1   8 KB873333                       {Installed, Comments, Backup Dir, Fix...
  1   8 KB873339                       {Installed, Comments, Backup Dir, Fix...
  1   8 KB883939                       {Installed, Comments, Backup Dir, Fix...
  1   8 KB885250                       {Installed, Comments, Backup Dir, Fix...
```

3. To retrieve information on a single hotfix, you will need to add a *Where-Object* cmdlet. You can do this by using the up arrow to retrieve the previous *gci -path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\HotFix'* command and pipelining the resulting object into the *Where-Object* cmdlet. Supply a value for the name property, as seen in the code listed here. Alternatively, supply a "KB" number from the previous output.

```
GCI -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\HotFix' | where
{$_.Name -like "*KB928388"}
```

4. This concludes this procedure.

Understanding the Variable Provider

The variable provider provides access to the variables that are defined within Windows PowerShell. These variables include both user-defined variables, such as *$mred*, and system-defined variables, such as *$host*. You can obtain a listing of the cmdlets designed to work specifically with variables by using the *Get-Help* cmdlet and specifying the asterisk (*) variable. The commands used in the procedure are in the UnderstandingTheVariableProvider.txt and WorkingWithVariables.txt files. To return only cmdlets, we use the *Where-Object* cmdlet and filter on the category that is equal to cmdlet. This command is shown here:

```
Get-Help *variable | Where-Object {$_.category -eq ÒcmdletÓ}
```

The resulting list contains five cmdlets but is a little jumbled and difficult to read. So let's modify the preceding command and specify the properties to return. To do this, use the up arrow and pipeline the returned object into the *Format-List* cmdlet. Add the three properties we are interested in: name, category, and synopsis. The revised command is shown here:

```
Get-Help *variable | Where-Object {$_.category -eq "cmdlet"} |
Format-List name, category, synopsis
```

The resulting output is much easier to read and understand. It is shown here:

```
Name     : Get-Variable
Category : Cmdlet
Synopsis : Gets the variables in the current console.

Name     : New-Variable
Category : Cmdlet
Synopsis : Creates a new variable.

Name     : Set-Variable
Category : Cmdlet
Synopsis : Sets the value of a variable. Creates the variable if one with the requested
name does not exist.

Name     : Remove-Variable
Category : Cmdlet
Synopsis : Deletes a variable and its value.

Name     : Clear-Variable
Category : Cmdlet
Synopsis : Deletes the value of a variable.
```

Working with variables

1. Open Windows PowerShell.
2. Use the *Set-Location* cmdlet to set the working location to the variable PSDrive. Use the *sl* alias to reduce typing needs. This command is shown here:

   ```
   SL variable:\
   ```

3. Produce a complete listing of all the variables currently defined in Windows PowerShell. To do this, use the *Get-ChildItem* cmdlet. You can use the alias *gci* to produce this list. The command is shown here:

```
Get-ChildItem
```

4. The resulting list is jumbled. Use the up arrow to retrieve the *Get-ChildItem* command, and pipeline the resulting object into the *Sort-Object* cmdlet. Sort on the name property. This command is shown here:

```
Get-ChildItem | Sort {$_.Name}
```

5. The output from the previous command is shown here:

```
Name                           Value
----                           -----
$                              }
?                              True
^                              Get-ChildItem
_
args                           {}
ConfirmPreference              High
ConsoleFileName
DebugPreference                SilentlyContinue
Error                          {System.Management.Automation.ParseException:...
ErrorActionPreference          Continue
ErrorView                      NormalView
ExecutionContext               System.Management.Automation.EngineIntrinsics
false                          False
FormatEnumerationLimit         4
HOME                           C:\Documents and Settings\edwils.NORTHAMERICA
Host                           System.Management.Automation.Internal.Host.In...
input                          System.Array+SZArrayEnumerator
LASTEXITCODE                   0
lastWord                       get-c
line                           get-c
MaximumAliasCount              4096
MaximumDriveCount              4096
MaximumErrorCount              256
MaximumFunctionCount           4096
MaximumHistoryCount            64
MaximumVariableCount           4096
mred                           mred
MyInvocation                   System.Management.Automation.InvocationInfo
NestedPromptLevel              0
null
OutputEncoding                 System.Text.ASCIIEncoding
PID                            292
PROFILE                        C:\Documents and Settings\edwils.NORTHAMERICA...
ProgressPreference             Continue
PSHOME                         C:\WINDOWS\system32\WindowsPowerShell\v1.0
PWD                            Variable:\
ReportErrorShowExceptionClass  0
ReportErrorShowInnerException  0
ReportErrorShowSource          1
```

```
ReportErrorShowStackTrace      0
ShellId                        Microsoft.PowerShell
StackTrace                        at System.Number.StringToNumber(String str...
true                           True
VerbosePreference              SilentlyContinue
WarningPreference              Continue
WhatIfPreference               0
```

6. Use the *Get-Variable* cmdlet to retrieve a specific variable. Use the *ShellId* variable. You can use *Tab completion* to speed up typing. The command is shown here:

```
Get-Variable ShellId
```

7. Use the up arrow to retrieve the previous *Get-Variable ShellId* command. Pipeline the object returned into a *Format-List* cmdlet and return all properties. This is shown here:

```
Get-Variable ShellId | Format-List *
```

8. The resulting output includes the description of the variable, value, and other information shown here:

```
Name        : ShellId
Description : The ShellID identifies the current shell.  This is used by #Requires.
Value       : Microsoft.PowerShell
Options     : Constant, AllScope
Attributes  : {}
```

9. Create a new variable called *administrator*. To do this, use the *New-Variable* cmdlet. This command is shown here:

```
New-Variable administrator
```

10. Use the *Get-Variable* cmdlet to retrieve the new administrator variable. This command is shown here:

```
Get-Variable administrator
```

11. The resulting output is shown here. Notice that there is no value for the variable.

```
Name                           Value
----                           -----
administrator
```

12. Assign a value to the new administrator variable. To do this, use the *Set-Variable* cmdlet. Specify the *administrator* variable name, and supply your given name as the value for the variable. This command is shown here:

```
Set-Variable administrator -value mred
```

13. Use the up arrow one time to retrieve the previous *Get-Variable administrator* command. This command is shown here:

```
Get-Variable administrator
```

14. The output displays both the variable name and the value associated with the variable. This is shown here:

```
Name                           Value
----                           -----
administrator                  mred
```

15. Use the *Remove-Variable* cmdlet to remove the administrator variable you previously created. This command is shown here:

```
Remove-Variable administrator
```

16. Use the up arrow one time to retrieve the previous *Get-Variable administrator* command. This command is shown here:

```
Get-Variable administrator
```

17. The variable has been deleted. The resulting output is shown here:

```
Get-Variable : Cannot find a variable with name 'administrator'.
At line:1 char:13
+ Get-Variable <<<< administrator
```

18. This concludes this procedure.

Exploring the Certificate Provider: Step-by-Step Exercises

In this exercise, we explore the use of the Certificate provider in Windows PowerShell.

1. Start Windows PowerShell.
2. Obtain a listing of all the properties available for use with the *Get-ChildItem* cmdlet by piping the results into the *Get-Member* cmdlet. To filter out only the properties, pipeline the results into a *Where-Object* cmdlet and specify the *membertype* to be equal to property. This command is shown here:

```
Get-ChildItem |Get-Member | Where-Object {$_.membertype -eq "property"}
```

3. Set your location to the certificate drive. To identify the certificate drive, use the *Get-PSDrive* cmdlet. Use the *Where-Object* cmdlet and filter on names that begin with the letter c. This is shown here:

```
Get-PSDrive |where {$_.name -like "c*"}
```

4. The results of this command are shown here:

```
Name       Provider      Root                                    CurrentLocation
----       --------      ----                                    ---------------
C          FileSystem    C:\
cert       Certificate   \
```

5. Use the *Set-Location* cmdlet to change to the certificate drive.

```
Sl cert:\
```

6. Use the *Get-ChildItem* cmdlet to produce a listing of all the certificates on the machine.

```
GCI
```

7. The output from the previous command is shown here:

```
Location   : CurrentUser
StoreNames : {?, UserDS, AuthRoot, CA...}

Location   : LocalMachine
StoreNames : {?, AuthRoot, CA, AddressBook...}
```

8. The listing seems somewhat incomplete. To determine whether there are additional certificates installed on the machine, use the *Get-ChildItem* cmdlet again, but this time specify the recurse argument. Modify the previous command by using the up arrow. The command is shown here:

```
GCI -recurse
```

9. The output from the previous command seems to take a long time to run and produces hundreds of lines of output. To make the listing more readable, pipe the output to a text file, and then open the file in Notepad. The command to do this is shown here:

```
GCI -recurse >C:\a.txt;notepad.exe a.txt
```

10. This concludes this step-by-step exercise.

One Step Further: Examining the Environment Provider

In this exercise, we work with the Windows PowerShell Environment provider.

1. Start Windows PowerShell.
2. Use the *New-PSDrive* cmdlet to create a drive mapping to the alias provider. The name of the new PSDrive will be *al*. The PSProvider is alias, and the root will be dot (.). This command is shown here:

```
new-PSDrive -name al -PSProvider alias -Root .
```

3. Change your working location to the new PSDrive you called *al*. To do this, use the *sl* alias for the *Set-Location* cmdlet. This is shown here:

```
SL al:\
```

4. Use the *gci* alias for the *Get-ChildItem* cmdlet, and pipeline the resulting object into the *Sort-Object* cmdlet by using the *sort* alias. Supply name as the property to sort on. This command is shown here:

```
GCI | Sort -Property name
```

5. Use the up arrow to retrieve the previous *gci | sort -property name* command and modify it to use a *Where-Object* cmdlet to return aliases only when the name is greater than the letter t. Use the *where* alias to avoid typing the entire name of the cmdlet. The resulting command is shown here:

```
GCI | sort -Property name | Where {$_.Name -gt "t"}c
```

6. Change your location back to the C:\ drive. To do this, use the *sl* alias and supply the C:\ argument. This is shown here:

```
SL C:\
```

7. Remove the PSDrive mapping for al. To do this, use the *Remove-PSDrive* cmdlet and supply the name of the PSDrive to remove. Note, this command does not want a trailing colon (:) or colon with backslash (:\). The command is shown here:

```
Remove-PSDrive al
```

8. Use the *Get-PSDrive* cmdlet to ensure the al drive was removed. This is shown here:

```
Get-PSDrive
```

9. Use the *Get-Item* cmdlet to obtain a listing of all the environment variables. Use the path argument and supply env:\ as the value. This is shown here:

```
Get-Item -Path env:\
```

10. Use the up arrow to retrieve the previous command, and pipeline the resulting object into the *Get-Member* cmdlet. This is shown here:

```
Get-Item -Path env:\ | Get-Member
```

11. The results from the previous command are shown here:

```
   TypeName: System.Collections.Generic.Dictionary'2+ValueCollection[[System.St
ring, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e0
89],[System.Collections.DictionaryEntry, mscorlib, Version=2.0.0.0, Culture=neu
tral, PublicKeyToken=b77a5c561934e089]]

Name          MemberType   Definition
----          ----------   ----------
CopyTo        Method       System.Void CopyTo(DictionaryEntry[] array, Int32...
Equals        Method       System.Boolean Equals(Object obj)
GetEnumerator Method       System.Collections.Generic.Dictionary`2+ValueColl...
GetHashCode   Method       System.Int32 GetHashCode()
GetType       Method       System.Type GetType()
get_Count     Method       System.Int32 get_Count()
ToString      Method       System.String ToString()
PSDrive       NoteProperty System.Management.Automation.PSDriveInfo PSDrive=Env
PSIsContainer NoteProperty System.Boolean PSIsContainer=True
PSPath        NoteProperty System.String PSPath=Microsoft.PowerShell.Core\En...
PSProvider    NoteProperty System.Management.Automation.ProviderInfo PSProvi...
Count         Property     System.Int32 Count {get;}
```

12. Press the up arrow twice to return to the *get-item -path env:* command. Use the Home key to move your insertion point to the beginning of the line. Add a variable called *$objEnv* and use it to hold the object returned by the *get-item -path env:* command. The completed command is shown here:

```
$objEnv=Get-Item -Path env:\
```

13. From the listing of members of the environment object, find the count property. Use this property to print out the total number of environment variables. As you type **$o**, try to use *Tab completion* to avoid typing. Also try to use *Tab completion* as you type the *c* in count. The completed command is shown here:

```
$objEnv.Count
```

14. Examine the methods of the object returned by *get-item -path env:*. Notice there is a Get_Count method. Let's use that method. The code is shown here:

```
$objEnv.Get_count
```

15. When this code is executed, however, the results define the method rather than execute the Get_Count method. These results are shown here:

```
MemberType          : Method
OverloadDefinitions : {System.Int32 get_Count()}
TypeNameOfValue     : System.Management.Automation.PSMethod
Value               : System.Int32 get_Count()
Name                : get_Count
IsInstance          : True
```

16. To retrieve the actual number of environment variables, we need to use empty parentheses is at the end of the method. This is shown here:

```
$objEnv.Get_count()
```

17. If you want to know exactly what type of object you have contained in the *$objEnv* variable, you can use the GetType method, as shown here:

```
$objEnv.GetType()
```

18. This command returns the results shown here:

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
False    True     ValueCollection                          System.Object
```

19. This concludes this one step further exercise.

Chapter 3 Quick Reference

To	Do This
Produce a listing of all variables defined in a Windows PowerShell session	Use the *Set-Location* cmdlet to change location to the variable PSDrive, then use the *Get-ChildItem* cmdlet
Obtain a listing of all the aliases	Use the *Set-Location* cmdlet to change location to the alias PSDrive, then use the *Get-ChildItem* cmdlet to produce a listing of aliases. Pipeline the resulting object into the *Where-Object* cmdlet and filter on the name property for the appropriate value
Delete a directory that is empty	Use the *Remove-Item* cmdlet and supply the name of the directory
Delete a directory that contains other items	Use the *Remove-Item* cmdlet and supply the name of the directory and specify the recurse argument
Create a new text file	Use the *New-Item* cmdlet and specify the -path argument for the directory location. Supply the name argument, and specify the type argument as file. Example: *new-item -path C:\Mytest -name Myfile.txt -type file*
Obtain a listing of registry keys from a registry hive	Use the *Get-ChildItem* cmdlet and specify the appropriate PSDrive name for the -path argument. Complete the path with the appropriate registry path. Example: *gci -path HKLM:\software*
Obtain a listing of all functions on the system	Use the *Get-ChildItem* cmdlet and supply the PSDrive name of *function:* to the path argument. Example: *gci -path function:*

Chapter 8

Leveraging the Power of ADO

After completing this chapter, you will be able to:

- Understand the use of ADO in Windows PowerShell scripts
- Connect to Active Directory to perform a search
- Control the way data are returned
- Use compound query filters

Connecting to Active Directory with ADO

In this section, you will learn a special query technique to search Active Directory using ActiveX Data Objects (ADO). The technique is exactly the same technique you will use to search other databases. You will be able to use the results returned by that custom query to perform additional tasks. For example, you could search Active Directory for all users who don't have telephone numbers assigned to them. You could then send that list to the person in charge of maintaining the telephone numbers. Even better, you could modify the search so that it returns the user names and their managers' names. You could then take the list of users with no phone numbers that is returned and send e-mail to the managers to update the phone list in Active Directory. The functionality incorporated in your scripts is primarily limited by your imagination. The following list summarizes uses of the search technology:

- Query Active Directory for a list of computers that meet a given search criterion
- Query Active Directory for a list of users who meet a given search criterion
- Query Active Directory for a list of printers that meet a given search criterion
- Use the data returned from the preceding three queries to perform additional operations

All the scripts mentioned in this chapter can be found in the corresponding scripts folder on the CD.

Just the Steps To search Active Directory

1. Create a connection to Active Directory by using ADO.
2. Use the Open() method of the object to access Active Directory.
3. Create an ADO Command object and assign the ActiveConnection property to the Connection object.
4. Assign the query string to the CommandText property of the Command object.
5. Use the Execute() method to run the query and store the results in a RecordSet object.
6. Read information in the result set using properties of the RecordSet object.
7. Close the connection by using the Close() method of the Connection object.

The script BasicQuery.ps1 (shown later) illustrates how to search Active Directory by using ADO. Keep in mind that BasicQuery.ps1 can be used as a template script to make it easy to perform Active Directory queries using ADO.

The BasicQuery.ps1 script begins with defining the query that will be used. The string is stored in the *$strQuery* variable. When querying Active Directory using ADO, there are two ways the query can be specified. The one used here is called the *Lightweight Directory Access Protocol (LDAP) dialect*. The other means of specifying the query is called the *SQL dialect* and will be explored later in this chapter.

The LDAP dialect string is made up of four parts. Each of the parts is separated by a semicolon. If one part is left out, then the semicolon must still be present. This is actually seen in the BasicQuery.ps1 script because we do not supply a value for the filter portion. This line of code is shown here:

```
$strQuery = "<LDAP://dc=nwtraders,dc=msft>;;name;subtree"
```

Table 8-1 illustrates the LDAP dialect parts. In the BasicQuery.ps1 script, the filter is left out of the query. The base portion is used to specify the exact point of the connection into Active Directory. Here we are connecting to the root of the NwTraders.msft domain. We could connect to an organizational unit (OU) called MyTestOU by using the distinguished name, as shown here:

```
ou=myTestOU,dc=nwtraders,dc=msft
```

Table 8-1 LDAP Dialect Query Syntax

Base	Filter	Attributes	Search Scope
<LDAP://dc=nwtraders,dc=msft>	(objectCategory=computer)	name	subtree

When we create the filter portion of the LDAP dialect query, we specify the attribute name on the left and the value for the attribute we are looking for on the right. If I were looking for every object that had a location of Atlanta, then the filter would look like the one shown here:

```
(l=Atlanta)
```

The attribute portion of the LDAP query is a simple list of attributes you are looking for, each separated by a comma. If after you had found objects in Atlanta, you wanted to know the name and category of the objects, your attribute list would look like the following:

```
Name, objectCategory
```

The search scope is the last portion of the LDAP dialect query. There are three possible values for the search scope. The first is base. If we specify the search scope as base, then it will only return the single that was the target of the query, that is, the base portion of the query. Using base is valuable if you want to determine whether an object is present in active directory.

The second allowable value for the search scope is oneLevel. When you use the search scope of oneLevel, it will return the Child objects of the base of your query. It does not, however, perform a recursive query. If your base is an OU, then it will list the items contained in the OU. But it will not go into any child OUs and list their members. This is an effective query technique and should be considered standard practice.

The last allowable value for the search scope is subtree. Subtree begins at the base of your query and then recurses into everything under the base of your query. It is sometimes referred to in the Platform Software Development Kit (SDK) as the deep search option because it will dig deeply into all sublevels of your Active Directory hierarchy. If you target the domain root, then it will go into every OU under the domain root, and then into the child OUs, and so forth. This should be done with great care because it can generate a great deal of network traffic and a great deal of workload on the server. If you do need to perform such a query, then you should perform the query asynchronously, and use paging to break the result set into smaller chunks. This will level out the network utilization. In addition, you should try to include one attribute that is indexed. If the attributes you are interested in are replicated to the Global Catalog (GC), then you should query the GC instead of connecting to rootDSE (DSA-specific entry). These techniques will all be examined in this chapter.

After the query is defined, we need to create two objects. We will use the *New-Object* cmdlet to create these objects. The first object to create is the ADODB.Connection object. The line of code used to create the Connection object is shown here:

```
$objConnection = New-Object -comObject "ADODB.Connection"
```

This object is a com object and is contained in the variable *$objConnection*. The second object that is needed is the ADODB.Command object. The code to create the Command object is shown here:

```
objCommand = New-Object -comObject "ADODB.Command"
```

After the two objects are created, we need to open the connection into Active Directory. To open the connection, we use the Open method from the ADODB.Connection object. When we call the Open method, we need to specify the name of the provider that knows how to read the Active Directory database. For this, we will use the ADsDSOObject provider. This line of code is shown here:

```
$objConnection.Open("Provider=ADsDSOObject;")
```

After the connection into the Active Directory database has been opened, we need to associate the Command object with the Connection object. To do this, we use the ActiveConnection property of the Command object. The line of code that does this is shown here:

```
$objCommand.ActiveConnection = $objConnection
```

Now that we have an active connection into Active Directory, we can go ahead and assign the query to the command text of the Command object. To do this, we use the CommandText property of the Command object. In the BasicQuery.ps1 script, we use the following line of code to do this:

```
$objCommand.CommandText = $strQuery
```

After everything is lined up, we call the Execute method of the Command object. The Execute method will return a RecordSet object, which is stored in the *$objRecordSet* variable. This line of code is shown here:

```
$objRecordSet = $objCommand.Execute()
```

To examine individual records from the RecordSet object, we use the *do ... until* statement to walk through the collection. The script block of the *do ... until* statement is used to retrieve the Name property from the RecordSet object. To retrieve the specific property, we retrieve the Fields.Item property and specify the property we retrieved from the attributes portion of the query. We then pipeline the resulting object into the *Select-Object* cmdlet and choose both the name and the Value property. This line of code is shown here:

```
$objRecordSet.Fields.item("name") |Select-Object Name,Value
```

To move to the next record in the recordset, we need to use the MoveNext method from the RecordSet object. This line of code is shown here:

```
$objRecordSet.MoveNext()
```

The complete BasicQuery.ps1 script is shown here:

BasicQuery.ps1

```
$strQuery = "<LDAP://dc=nwtraders,dc=msft>;;name;subtree"

$objConnection = New-Object -comObject "ADODB.Connection"
$objCommand = New-Object -comObject "ADODB.Command"
$objConnection.Open("Provider=ADsDSOObject;")
$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
```

```
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") |Select-Object Name,Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()
```

Quick Check

Q. What technology is utilized to search Active Directory?

A. DO is the technology that is used to search Active Directory.

Q. Which part of the script is used to perform the query?

A. The command portion of the script is used to perform the query.

Q. How are results returned from an ADO search of Active Directory?

A. The results are returned in a recordset.

Creating More Effective Queries

The BasicQuery.ps1 script is a fairly wasteful script in that all it does is produce a list of user names and print them out. Although the script illustrates the basics of making a connection into Active Directory by using ADO, it is not exactly a paradigm of efficiency. ADO, however, is a very powerful technology, and there are many pieces of the puzzle we can use to make the script more efficient and more effective. The first thing we need to do is to understand the objects we have that we can use with ADO. These objects are listed in Table 8-2.

Table 8-2 Objects Used to Search Active Directory

Object	Description
Connection	An open connection to an OLE DB data source such as ADSI
Command	Defines a specific command to execute against the data source
Parameter	An optional collection for any parameters to provide to the Command object
RecordSet	A set of records from a table, a Command object, or SQL syntax A RecordSet object can be created without any underlying Connection object
Field	A single column of data in a recordset
Property	A collection of values supplied by the provider for ADO
Error	Contains details about data access errors. Refreshed when an error occurs in a single operation

When we use ADO to talk to Active Directory, we often are working with three different objects: the Connection object, the Command object, and the RecordSet object. The Command object is used to maintain the connection, pass along the query parameters, and perform such tasks as specifying the page size and search scope and executing the query. The Connection object is used to load the provider and to validate the user's credentials. By default, it utilizes the credentials of the currently logged-on user. If you need to specify alternative credentials, you can use the properties listed in Table 8-3. To do this, we need to use the Properties property of the Connection object. After we have the Connection object, and we use Properties to get to the properties, we then need to use Item to supply value for the specific property item we want to work with.

Table 8-3 Authentication Properties for the Connection Object

Property	Description
User ID	A string that identifies the user whose security context is used when performing the search. (For more information about the format of the user name string, see IADsOpenDSObject::OpenDSObject in the Platform SDK.) If the value is not specified, the default is the logged-on user or the user impersonated by the calling process.
Password	A string that specifies the password of the user identified by "User ID"
Encrypt Password	A Boolean value that specifies whether the password is encrypted. The default is False.
ADSI Flag	A set of flags from the ADS_AUTHENTICATION_ENUM enumeration. The flag specifies the binding authentication options. The default is zero.

Using Alternative Credentials

As network administration becomes more granular, with multiple domains, work groups, OUs, and similar grouping techniques, it becomes less common for everyone on the IT team to be a member of the Domain Admins group. If the script does not impersonate a user who is a member of the Domain Admins group, then it is quite likely it will need to derive permissions from some other source. One method to do this is to supply alternative credentials in the script. To do this, we need to specify certain properties of the Connection object.

Just the Steps To create a connection in Active Directory using alternative credentials

1. Create the ADODB.Connection object
2. Use the Provider property to specify the ADsDSOObject provider
3. Use Item to supply a value for the properties "User ID" and "Password"
4. Open the connection while supplying a value for the name of the connection

The technique outlined in the using alternative credentials step-by-step exercise is shown here. This code is from the QueryComputersUseCredentials.ps1 script, which is developed in the querying active directory using alternative credentials procedure.

```
$objConnection = New-Object -comObject "ADODB.Connection"
$objConnection.provider = "ADsDSOObject"
$objConnection.properties.item("user ID") = $strUser
$objConnection.properties.item("Password") = $strPwd
$objConnection.open("modifiedConnection")
```

Querying Active Directory using alternative credentials

1. Open the QueryComputers.ps1 script in Notepad or in your favorite Windows PowerShell script editor and save it as *yourname*QueryComputersUseCredentials.ps1.
2. On the first noncommented line, define a new variable called *$strBase*, and use it to assign the LDAP connection string. This variable is used to define the base of the query into Active Directory. For this example, we will connect to the root of the NwTraders.msft domain. To do this, the string is enclosed in angle brackets and begins with the moniker LDAP. The base string is shown here:

   ```
   "<LDAP://dc=nwtraders,dc=msft>"
   ```

   The new line of code is shown here:

   ```
   $strBase = "<LDAP://dc=nwtraders,dc=msft>"
   ```

3. Create a new variable called *$strFilter*. This will be used to hold the filter portion of our LDAP syntax query. Assign a string that specifies the objectCategory attribute when it is equal to the value of computer. This is shown here:

   ```
   $strFilter = "(objectCategory=computer)"
   ```

4. Create a new variable called *$strAttributes* and assign the string of name to it. This variable will be used to hold the attributes to search on in Active Directory. This line of code is shown here:

   ```
   $strAttributes = "name"
   ```

5. Create a variable called *$strScope*. This variable will be used to hold the search scope parameter of our LDAP syntax query. Assign the value of subtree to it. This line of code is shown here:

   ```
   $strScope = "subtree"
   ```

6. Modify the *$strQuery* line of code so that it uses the four variables we created:

   ```
   $strQuery = "<LDAP://dc=nwtraders,dc=msft>;;name;subtree"
   ```

   The advantage of this is that each of the four parameters that are specified for the LDAP syntax query can easily be modified by simply changing the value of the variable. This preserves the integrity of the worker section of the script. The order of the four

parameters is base, filter, attributes, and scope. Thus, the revised value to assign to the *$strQuery* variable is shown here:

```
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
```

7. Create a new variable called *$strUser* and assign the string "LondonAdmin" to it. This is the name of the useraccount to use to make the connection to Active Directory. This line of code is shown here:

```
$strUser = "LondonAdmin"
```

8. Create a new variable called *$strPassword* and assign the string Password1 to it. This is the password that will be used when connecting into the NwTraders.msft domain by using the LondonAdmin account. This is shown here:

```
$strPwd = "Password1"
```

9. Between *the $objConnection = New-Object -comObject "ADODB.Connection"* command and the *$objCommand = New-Object -comObject "ADODB.Command"* command, insert four blank lines. This space will be used for rearranging the code and for inserting new properties on the Connection object. The revised code is shown here:

```
$objConnection = New-Object -comObject "ADODB.Connection"



$objCommand = New-Object -comObject "ADODB.Command"
```

10. Move the $objConnection.provider = "ADsDSOObject;" line of code from its position below the $objCommand = New-Object −comObject "ADODB.Command" line of code to below the line of code that creates the Connection object. After you have the code moved, remove the trailing semicolon because it is not needed. This revised code is shown here:

```
$objConnection = New-Object -comObject "ADODB.Connection"
$objConnection.provider = "ADsDSOObject"
```

11. Use the Item method of the properties collection of the Connection object to assign the value contained in the *$strUser* variable to the "User ID" property. This line of code is shown here:

```
$objConnection.properties.item("user ID") = $strUser
```

12. Use the Item method of the properties collection of the Connection object to assign the value contained in the *$strPassword* variable to the "Password" property. This line of code is shown here:

```
$objConnection.properties.item("Password") = $strPwd
```

13. The last line of code we need to modify from the old script is the $objConnection.Open("Provider=ADsDSOObject;") line. Because we needed to move the provider string up earlier in the code to enable us to modify the properties, we have already

specified the provider. So, now we only need to open the connection. When we open the connection, we give it a name "modifiedConnection" that we would be able to use later on in the script if we so desired. The revised line of code is shown here:

```
$objConnection.open("modifiedConnection")
```

14. Save and run your script. If it does not perform as expected, compare it with the QueryComputersUseCredentials.ps1 script.

15. This concludes the querying Active Directory using alternative credentials procedure.

Modifying Search Parameters

A number of search options are available to the network administrator. The use of these search options will have an extremely large impact on the performance of your queries against Active Directory. It is imperative, therefore, that you learn to use the following options. Obviously, not all options need to be specified in each situation. In fact, in many situations, the defaults will perform just fine. However, if a query is taking a long time to complete, or you seem to be flooding the network with unexpected traffic, you might want to take a look at the Search properties in Table 8-4.

Table 8-4 ADO Search Properties for Command Object

Property	Description
Asynchronous	A Boolean value that specifies whether the search is synchronous or asynchronous. The default is False (synchronous). A synchronous search blocks until the server returns the entire result (or for a paged search, the entire page). An asynchronous search blocks until one row of the search results is available, or until the time specified by the Timeout property elapses.
Cache Results	A Boolean value that specifies whether the result should be cached on the client side. The default is True; ADSI caches the resultset. Turning off this option might be desirable for large resultsets.
Chase Referrals	A value from ADS CHASE_REFERRALS_ENUM that specifies how the search chases referrals. The default is ADS_CHASE_REFERRALS EXTERNAL = 0x40. To set ADS_CHASE_REFERRALS_NEVER, set to 0.
Column Names Only	A Boolean value that indicates that the search should retrieve only the name of attributes to which values have been assigned. The default is False.
Deref (dereference) Aliases	A Boolean value that specifies whether aliases of found objects are resolved. The default is False.
PageSize	An integer value that turns on paging and specifies the maximum number of objects to return in a resultset. The default is no page size, which means that after 1000 items have been delivered from Active Directory, that is it. To turn on paging, you must supply a value for page size, and it must be less than the SizeLimit property. (For more information, see PageSize in the Platform SDK, which is available online from *http://msdn2.microsoft.com/*.)

Table 8-4 ADO Search Properties for Command Object

Property	Description
SearchScope	A value from the ADS_SCOPEENUM enumeration that specifies the search scope. The default is ADS_SCOPE_SUBTREE.
SizeLimit	An integer value that specifies the size limit for the search. For Active Directory, the size limit specifies the maximum number of returned objects. The server stops searching once the size limit is reached and returns the results accumulated up to that point. The default is No Limit.
Sort on	A string that specifies a comma-separated list of attributes to use as sort keys. This property works only for directory servers that support the LDAP control for server-side sorting. Active Directory supports the sort control, but this control can affect server performance, particularly when the resultset is large. Be aware that Active Directory supports only a single sort key. The default is No Sorting.
TimeLimit	An integer value that specifies the time limit, in seconds, for the search. When the time limit is reached, the server stops searching and returns the results accumulated to that point. The default is No Time Limit.
Timeout	An integer value that specifies the client-side timeout value, in seconds. This value indicates the time the client waits for results from the server before quitting the search. The default is No Timeout.

In the previous section, when we used alternative credentials in the script, we specified various properties on the Connection object. We will use the same type of procedure to modify search parameters, but this time we will specify values for various properties on the Command object. As an example, suppose we wanted to perform an asychronous query of Active Directory. We would need to supply a value of true for the asynchronous property. The technique is exactly the same as supplying alternative credentials: create the object, and use the Item method to specify a value for the appropriate property. This piece of code, taken from the AsynchronousQueryComputers.ps1 script, is shown here:

```
$objCommand = New-Object -comObject "ADODB.Command"
$objCommand.ActiveConnection = $objConnection
$objCommand.Properties.item("Asynchronous") = $blnTrue
```

Important When specifying properties for the Command object, ensure you have the activeConnection associated with a valid Connection object before making the assignment. Otherwise, you will get an error stating the property is not valid—which can be very misleading.

Note that you should specify a page size. In Windows Server 2003, Active Directory is limited to returning 1000 objects from the results of a query when no page size is specified. The PageSize property tells Active Directory how many objects to return at a time. When this property is specified, there is no limit on the number of returned objects Active Directory can provide. If you specify a size limit, the page size must be smaller. The exception would be if you

want an unlimited size limit of 0, then obviously the PageSize property would be larger than the value of 0. In the SizeLimitQueryUsers.ps1 script, after creating a Command object, associating the connection with the activeConnection property, we use the Item method of the properties collection to specify a size limit of 4. When the script is run, it only returns four users. The applicable portion of the code is shown here:

```
$objCommand = New-Object -comObject "ADODB.Command"
$objCommand.ActiveConnection = $objConnection
$objCommand.Properties.item("Size Limit") = 4
```

Controlling script execution

1. Open the QueryComputersUseCredentials.ps1 script, and save it as *yourname*QueryTimeOut.ps1.
2. Edit the query filter contained in the string that is assigned to the variable *$strFilter* so that the filter will return items when the ObjectCategory is equal to User. The revised line of code is shown here:

   ```
   $strFilter = "(objectCategory=User)"
   ```

3. Delete the line that creates the *$strUser* variable and assigns the LondonAdmin User to it.
4. Delete the line that creates the *$strPwd* variable and assigns the string Password1 to it.
5. Delete the two lines of code that assign the value contained in the *$strUser* variable to the User ID property, and the one that assigns the value contained in the *$strPwd* variable to the Password property of the Connection object. These two lines of code are shown commented-out here:

   ```
   #$objConnection.properties.item("user ID") = $strUser
   #$objConnection.properties.item("Password") = $strPwd
   ```

6. Inside the parentheses of the Open command that opens the Connection object to Active Directory, delete the Reference string that is contained inside it. We are able to delete this string because we did not use it to refer to the connection later in the script. The modified line of code is shown here:

   ```
   $objConnection.open()
   ```

7. Under the line of code that assigns the Connection object that is contained in the *$objConnection* variable to the ActiveConnection property of the Command object, we want to add the value of 1 to the TimeLimit property of the Command object. To do this, use the property name TimeLimit, and use the Item method to assign it to the properties collection of the Command object. The line of code that does this is shown here:

   ```
   $objCommand.properties.item("Time Limit")=1
   ```

8. Save and run your script. If it does not produce the desired results, compare it with QueryTimeOut.ps1.
9. This concludes the controlling script execution procedure.

Searching for Specific Types of Objects

One of the best ways to improve the performance of Active Directory searches is to limit the scope of the search operation. Fortunately, searching for a specific type of object is one of the easiest tasks to perform. For example, to perform a task on a group of computers, limit your search to the computer class of objects. To work with only groups, users, computers, or printers, specify the objectClass or the objectCategory attribute in the search filter. The objectCategory attribute is a single value that specifies the class from which the object in Active Directory is derived. In other words, users are derived from an objectCategory called *users*. All the properties you looked at in Chapter 7, "Working with Active Directory," when we were creating objects in Active Directory are contained in a template called an *objectCategory*. When you create a new user, Active Directory does a lookup to find out what properties the user class contains. Then it copies all those properties onto the new user you just created. In this way, all users have the same properties available to them.

Just the Steps To limit the Active Directory search

1. Create a connection to Active Directory by using ADO.
2. Use the Open method of the object to access Active Directory.
3. Create an ADO Command object, and assign the ActiveConnection property to the Connection object.
4. Assign the query string to the CommandText property of the Command object.
5. In the query string, specify the objectCategory of the target query.
6. Choose specific fields of data to return in response to the query.
7. Use the Execute method to run the query and store the results in a RecordSet object.
8. Read information in the result set using properties of the RecordSet object.
9. Close the connection by using the Close method of the Connection object.

In the QueryComputers.ps1 script, you use ADO to query Active Directory with the goal of returning a recordset containing selected properties from all the computers with accounts in the directory.

To make the script easier to edit, we abstracted each of the four parts of the LDAP dialect query into a separate variable. The *$strBase* variable in the QueryComputers.ps1 script is used to hold the base of the ADO query. The base is used to determine where the script will make its connection into Active Directory. The line of code that does this in the QueryComputers.ps1 script is shown here:

```
$strBase = "<LDAP://dc=nwtraders,dc=msft>"
```

The filter is used to remove the type of objects that are returned by the ADO query. In the QueryComputers.ps1 script, we filter on the value of the objectCategory attribute when it has a value of computer. This line of code is shown here:

```
$strFilter = "(objectCategory=computer)"
```

The attributes to be selected from the query are specified in the *$strAttributes* variable. In the QueryComputers.ps1 script, we choose only the Name attribute. This line of code is shown here:

```
$strAttributes = "name"
```

The search scope determines how deep the query will go. There are three possible values for this: base, oneLevel, and subtree. Base searches only at the level where the script connects. OneLevel tells ADO to go one level below where the *$strbase* connection is made. Subtree is probably the most commonly used and tells ADO to make a recursive query through Active Directory. This is the kind of query we do in QueryComputers.ps1. This line of code is shown here:

```
$strScope = "subtree"
```

The $strQuery is used to hold the query used to query from Active Directory. When it is abstracted into variables, it becomes easy to modify. The revised code is shown here:

```
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
```

The complete QueryComputers.ps1 is shown here:

QueryComputers.ps1

```
$strBase = "<LDAP://dc=nwtraders,dc=msft>"
$strFilter = "(objectCategory=computer)"
$strAttributes = "name"
$strScope = "subtree"
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"

$objConnection = New-Object -comObject "ADODB.Connection"
$objCommand = New-Object -comObject "ADODB.Command"
$objConnection.Open("Provider=ADsDSOObject;")
$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") |Select-Object Name,Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()
```

Querying multiple attributes

1. Open Notepad or your favorite Windows PowerShell editor.
2. Open QueryComputers.ps1, and save it as *yourname*QueryComputersByName.ps1.
3. Edit the $strFilter line so that it includes the additional attribute name. To do this using the LDAP dialect, we will need to first add an extra set of parentheses around the entire filter expression. This is shown here:

```
$strFilter = "((objectCategory=computer))"
```

4. Between the first set of double parentheses, we will add the ampersand character (&), which will tell the LDAP dialect search filter we want both of the attributes we are getting ready to supply. This is shown here:

```
$strFilter = "(&(objectCategory=computer))"
```

5. At end of the first search filter expression, we want to add a second expression. We want to search by both Computer Type objects and usernames. This modified line of code is shown here:

```
$strFilter = "(&(objectCategory=computer)(name=london))"
```

6. Save and run your script. It should produce a script output that lists all computer accounts named London.
7. This concludes the querying multiple attributes procedure.

What Is Global Catalog?

As you become more proficient in writing your scripts, and as you begin to work your magic on the enterprise on a global scale, you will begin to wonder why some queries seem to take forever and others run rather fast. After configuring some of the parameters you looked at earlier, you might begin to wonder whether you're hitting a Global Catalog (GC) server. A *Global Catalog server* is a server that contains all the objects and their associated attributes from your local domain. If all you have is a single domain, it doesn't matter whether you're connecting to a domain controller or a GC server because the information will be the same. If, however, you are in a multiple domain forest, you might very well be interested in which GC server you are hitting. Depending on your network topology, you could be executing a query that is going across a slow WAN link. You can control replication of attributes by selecting the Global Catalog check box. You can find this option by opening the Active Directory Schema MMC, highlighting the Attributes container. The Active Directory Schema MMC is not available by default in the Administrative Tools program group. For information on how to install it, visit the following URL: *http://technet2.microsoft.com/WindowsServer/en/library/2218144f-bb92-454e-9334-186ee7c740c61033.mspx?mfr=true*.

In addition to controlling the replication of attributes, the erstwhile administrator might also investigate attribute indexing (Fig. 8-1.) Active Directory already has indexes built on certain

objects. However, if an attribute is heavily searched on, you might consider an additional index. You should do this, however, with caution because an improperly placed index is worse than no index at all. The reason for this is the time spent building and maintaining an index. Both of these operations use processor time and disk I/O.

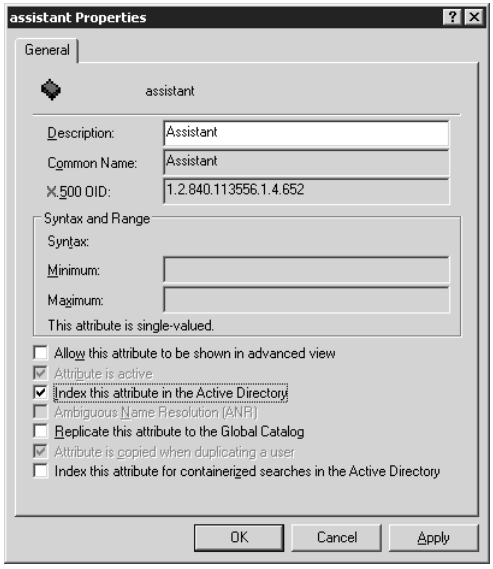

Figure 8-1 Heavily queried attributes often benefit from indexing

Querying a global catalog server

1. Open the BasicQuery.ps1 script in Notepad or another Windows PowerShell editor and save the file as *yourname*QueryGC.ps1

2. On the first noncommented line of your script, declare a variable called *$strBase*. This variable will be used to control the connection into the global catalog server in Active Directory. To do this, instead of using the LDAP moniker, we will use the GC moniker. The rest of the path will be the same because it uses the Distinguished Name of target. For this procedure, let's connect to the OU called MyTestOU in the NwTraders.msft domain. The line of code to do this is shown here:

```
$strBase = <GC://ou=MyTestOU,dc=nwtraders,dc=msft>
```

3. On the next line, create a variable called *$strFilter*. This variable will be used to hold the filter portion of the query. The filter will be used to return only objects from Active Directory that have the objectCategory attribute set to User. The line of code that does this is shown here:

```
$strFilter = "(objectCategory=user)"
```

4. Create a variable called *$strAttributes* that will be used to hold the attributes to retrieve from Active Directory. For this script, we are only interested in the Name attribute. The line of code that does this is shown here:

```
$strAttributes = "name"
```

5. Create a variable called *$strScope*. This variable will hold the string oneLevel that is used to tell Active Directory that we want the script to obtain a list of the users in the MyTestOU only. We do not need to perform a recursive type of query. This line of code is shown here:

```
$strScope = "oneLevel"
```

6. Modify the *$strQuery* line so that it uses the four variables we defined for each of the four parts of the LDAP dialect query. The four variables are *$strBase, $strFilter, $strAttributes,* and *$strScope* in this order. Make sure you use a semicolon to separate the four parts from one another. Move the completed line of code to the line immediately beneath the $strScope-"oneLevel" line. The completed line of code is shown here:

```
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
```

7. Save and run your script. It should run without errors. If it does not produce the expected results, compare your script with the QueryGC.ps1 script.
8. This concludes the querying a global catalog server procedure.

Querying a specific server

1. Open the QueryGC.ps1 script in Notepad or your favorite Windows PowerShell script editor, and save the script as *yourname*QuerySpecificServer.ps1.
2. Edit the string assigned to the *$strBase* variable so that you use the LDAP moniker instead of the GC moniker. After the ://, type the name of the server. Do not use CN=, as would normally be used for the Distinguished Name attribute. Instead, just type the name of the server followed by a forward slash (\). The completed line of code is shown here:

```
$strBase = "<LDAP://London/ou=MyTestOU,dc=nwtraders,dc=msft>"
```

3. Save and run your script. If your script does not work properly, compare it with the QuerySpecificServer.ps1 script.
4. This concludes the querying a specific server procedure.

Querying a specific server by IP address

1. Open the QuerySpecificServer.ps1 script in Notepad or your favorite Windows PowerShell script editor and save it as *yourname*QuerySpecificServerByIP.ps1.
2. Edit the string that is assigned to the *$strBase* variable so that you are supplying the LDAP moniker with an IP address instead of a Host name. The revised line of code is shown here:

```
$strBase = "<LDAP://192.168.1.1/ou=MyTestOU,dc=nwtraders,dc=msft>"
```

3. Save and run your script. If your script does not run properly, compare it with the QuerySpecificServerByIP.ps1 script.
4. This concludes the querying a specific server by IP address procedure.

Using the base search scope

1. Open the QuerySpecificServer.ps1 script in Notepad, or some other Windows PowerShell script editor. Save the script as *yourname*SearchBase.ps1.
2. Change the $strScope line so that it will point to base instead of oneLevel. This revised line of code is shown here:

```
$strScope = "base"
```

3. Because a base query connects to a specific object, there is no point in having a filter. Delete the $strFilter line, and remove the *$strFilter* variable from the second position of the *$strQuery* string that is used for the LDAP dialect query. The revised $strQuery line of code is shown here:

```
$strQuery = "$strBase;;$strAttributes;$strScope"
```

4. Because the base query will only return a single object, it does not make sense to perform a *do ... until* loop. Delete the line that has the opening *Do*, and delete the line with the *Until ($objRecordSet.eof)*.
5. Delete the opening and the closing curly brackets. Delete the *$objRecordSet.MoveNext()* command because there are no more records to move to.
6. Go to the *$strAttributes* variable and modify it so that we retrieve both the objectCategory and the Name attributes. The revised line of code is shown here:

```
$strAttributes = "objectCategory,name"
```

7. Copy the $objRecordSet.Fields.item("name") |Select-Object value line of code, and paste it just below the first one. Edit the first $objRecordSet.Fields.item line of code so that it will retrieve the objectCategory attribute from the recordset. The two lines of code are shown here:

```
$objRecordSet.Fields.item("objectCategory") |Select-Object value
$objRecordSet.Fields.item("name") |Select-Object value
```

8. Save and run your script. If it does not perform correctly, compare it with the Search Base.ps1 script.
9. This concludes the using the base search scope procedure.

Using the SQL Dialect to Query Active Directory

For many network professionals, the rather cryptic way of expressing the query into Active Directory is at once confusing and irritating. Because of this confusion, we also have an SQL dialect we can use to query Active Directory. The parts that make up an SQL dialect query are listed in Table 8-5. Of the four parts that can make up an SQL dialect query, only two parts are required: the *Select* statement and the *from* keyword that indicates the base for the search. An example of this use is shown here. A complete script that uses this type of query is the Select-NameSQL.ps1 script.

```
Select name from 'LDAP://ou=MyTestOu,dc=nwtraders,dc=msft'
```

Table 8-5 SQL Dialect

Select	From	Where	Order by
Comma separated list of attributes	AdsPath for the base of the search enclosed in single quotation marks	Optional Used for the filter	Optional. Used for server side sort control. A comma separated list of attributes

The *Where* statement is used to specify the filter for the Active Directory query. This is similar to the filter used in the LDAP dialect queries. The basic syntax of the filter is attributeName = value. But as in any SQL query, we are free to use various operators, as well as *and*, or *or*, and even wild cards. An example of a query using *Where* is shown here (keep in mind this is a single line of code that was wrapped for readability). A complete script that uses this type of query is the QueryComputersSQL.ps1 script.

```
Select name from 'LDAP://ou=MyTestOu,dc=nwtraders,dc=msft'where
objectCategory='computer'
```

The order by clause is the fourth part of the SQL dialect query. Just like the *Where* clause, it is also optional. In addition to selecting the property to order by, you can also specify two keywords: *ASC* for ascending and *DESC* for descending. An example of using the order by clause is shown here. A complete script using this query is the QueryUsersSQL.ps1 script.

```
Select adsPath, cn from 'LDAP://dc=nwtraders,dc=msft' where
objectCategory='user'order by sn DESC
```

The SQLDialectQuery.ps1 script is different from the BasicQuery.ps1 script only in the dialect used for the query language. The script still creates both a Connection object and a Command object, and works with a RecordSet object in the output portion of the script. In the SQLDialectQuery.ps1 script, we hold the SQL dialect query in three different variables. The *$strAttributes* variable holds the select portion of the script. *$strBase* is used to hold the AdsPath attribute, which contains the complete path to the target of operation. The last variable used in holding the query is the *$strFilter* variable, which holds the filter portion of the query. Using these different variables makes the script easier to modify and easier to read. The *$strQuery* variable is used to hold the entire SQL dialect query. If you are curious to see the

query put together in its entirety, you can simply print out the value of the variable by adding the $strQuery line under the line where the query is put back together.

SQLDialectQuery.ps1

```
$strAttributes = "Select name from "
$strBase = "'LDAP://ou=MyTestOu,dc=nwtraders,dc=msft'"
$strFilter = " where objectCategory='computer'"
$strQuery = "$strAttributes$strBase$strFilter"

$objConnection = New-Object -comObject "ADODB.Connection"
$objCommand = New-Object -comObject "ADODB.Command"
$objConnection.Open("Provider=ADsDSOObject;")
$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") |Select-Object Name,Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()
```

Creating an ADO Query into Active Directory: Step-by-Step Exercises

In this exercise, we will explore the use of various queries against Active Directory. We will use both simple and compound query filters as we return data, beginning with the generic and moving on to the more specific.

1. Launch the CreateMultipleUsers.ps1 script from the scripts folder for this chapter. This script will create 60 users with city locations from three different cities, and four different departments. We will use the different users and departments and cities in our Active Directory queries. By default, the script will create the users in the MyTestOU in the NwTraders.msft domain. If your Active Directory configuration is different, then edit the Active Directory Service Interfaces (ADSI) connection string shown here. If you are unsure of how to do this, refer back to Chapter 7, "Working with Active Directory."

   ```
   $objADSI = [ADSI]"LDAP://ou=myTestOU,dc=nwtraders,dc=msft"
   ```

2. Open Notepad or another Windows PowerShell script editor.
3. On the first line, declare a variable called *$strBase*. This variable will be used to hold the base for our LDAP syntax query into Active Directory. The string will use angle brackets at the beginning and the end of the string. We will be connecting to the MyTestOU in the NwTraders.msft domain. The line of code that does this is shown here:

   ```
   $strBase = "<LDAP://ou=mytestOU,dc=nwtraders,dc=msft>"
   ```

4. On the next line, declare a variable called *$strFilter*. This variable will hold the string that will be used for the query filter. It will filter out every object that is not a User object. The line of code that does this is shown here:

```
$strFilter = "(objectCategory=User)"
```

5. Create a variable called *$strAttributes*. This variable will hold the attribute we wish to retrieve from Active Directory. For this lab, we only want the name of the object. This line of code is shown here:

```
$strAttributes = "name"
```

6. On the next line, we need to declare a variable called *$strScope* that will hold the search scope parameter. For this exercise, we will use the subtree parameter. This line of code is shown here:

```
$strScope = "subtree"
```

7. On the next line, we put the four variables together to form our query string for the ADO query into Active Directory. Hold the completed string in a variable called *$strQuery*. Inside quotes, separate each of the four variables by a semicolon, which is used by the LDAP dialect to distinguish the four parts of the LDAP dialect query. The line of code to do this is shown here:

```
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
```

8. Create a variable called *$objConnection*. The *$objConnection* variable will be used to hold an ADODB.Connection COM object. To create the object, use the *New-Object* cmdlet. This line of code is shown here:

```
$objConnection = New-Object -comObject "ADODB.Connection"
```

9. Create a variable called *$objCommand* that will be used to hold a new instance of the COM object "ADODB.Command". The code to do this is shown here:

```
$objCommand = New-Object -comObject "ADODB.Command"
```

10. Open the Connection object by calling the Open method. Supply the name of the provider to use while opening the connection. For this lab, we will use the AdsDSOObject provider. The line of code that does is shown here:

```
$objConnection.Open("Provider=ADsDSOObject")
```

11. Now we need to associate the Connection object we just opened with the ActiveConnection property of the Command object. To do this, simply supply the Connection object contained in the *$objConnection* variable to the ActiveConnection property of the Command object. The code that does this is shown here:

```
$objCommand.ActiveConnection = $objConnection
```

12. Now we need to supply the text for the Command object. To do this, we will use the query contained in the variable *$strQuery* and assign it to the CommandText property of the Command object held in the *$objCommand* variable. The code that does this is shown here:

```
$objCommand.CommandText = $strQuery
```

13. It is time to execute the query. To do this, call the Execute method of the Command object. It will return a RecordSet object, so use the variable *$objRecordSet* to hold the RecordSet object that comes back from the query.

```
$objRecordSet = $objCommand.Execute()
```

14. Use a *do ... until* statement to walk through the recordset until you reach the end of file. While you are typing this, go ahead and open and close the curly brackets. This will take four lines of code, which are shown here:

```
Do
{


}
Until ($objRecordSet.eof)
```

15. Inside the curly brackets, retrieve the Name attribute from the recordset by using the Item method from the Fields property. Pipeline the resulting object into a *Select-Object* cmdlet and retrieve only the value property. This line of code is shown here:

```
$objRecordSet.Fields.item("name") |Select-Object Value
```

16. Call the MoveNext method to move to the next record in the RecordSet object contained in the *$objRecordSet* variable. This line of code is shown here:

```
$objRecordSet.MoveNext()
```

17. After the until ($objRecordSet.eof) line of code, call the Close method from the RecordSet object to close the connection into Active Directory. This line of code is shown here:

```
$objConnection.Close()
```

18. Save your script as *yourname*QueryUsersStepByStep.ps1. Run your script. You should see the name of 60 users come scrolling forth from the Windows PowerShell console. If this is not the case, compare your script with the QueryUsersStepByStep.ps1 script. Note, in the QueryUsersStepByStep.ps1 script, there are five $strFilter lines ... only one that is not commented out. This is so you will have documentation on the next steps. When this code is working, it is time to move on to a few more steps.

19. Now we want to modify the filter so that it will only return users who are in the Charlotte location. To do this, copy the $strFilter line and paste it below the current line of code. Now, comment out the original $strFilter. We now want to use a compound query: objects in Active Directory that are of the category user, and a location attribute of Charlotte. From Chapter 7, you may recall the attribute for location is l. To make a compound

query, enter the search parameter inside parentheses, inside the grouping parentheses, after the first search filter. This modified line of code is shown here:

```
$strFilter = "(&(objectCategory=User)(l=charlotte))"
```

20. Save and run your script. Now, we want to add an additional search parameter. Copy your modified $strFilter line, and paste it beneath the line you just finished working on. Comment out the previous $strFilter line. Just after the location filter of Charlotte, add a filter for only users in Charlotte who are in the HR department. This revised line of code is shown here:

```
$strFilter = "(&(objectCategory=User)(l=charlotte)(department=hr))"
```

21. Save and run your script. Now copy your previous $strFilter line of code, and paste it below the line you just modified. This change is easy. You want all users in Charlotte who are not in HR. To make a not query, place the exclamation mark (bang) operator inside the parentheses you wish the operator to affect. This modified line of code is shown here:

```
$strFilter = "(&(objectCategory=User)(l=charlotte)(!department=hr))"
```

22. Save and run your script. Because this is going so well, let's add one more parameter to our search filter. So, once again copy the $strFilter line of code you just modified, and paste it beneath the line you just finished working on. This time, we want users who are in Charlotte or Dallas and who are not in the HR department. To do this, add a l=dallas filter behind the l=charlotte filter. Put parentheses around the two locations, and then add the pipeline character (|) in front of the l=charlotte parameter. This revised line of code is shown here. Keep in mind that it is wrapped for readability, but should be on one logical line in the script.

```
$strFilter = "(&(objectCategory=User)(|(l=charlotte)(l=dallas))(!department=hr))"
```

23. Save and run your script. In case you were getting a little confused by all the copying and pasting, here are all the *$strFilter* commands you have typed in this section of the step-by-step exercise:

```
$strFilter = "(objectCategory=User)"
#$strFilter = "(&(objectCategory=User)(l=charlotte))"
#$strFilter = "(&(objectCategory=User)(l=charlotte)(department=hr))"
#$strFilter = "(&(objectCategory=User)(l=charlotte)(!department=hr))"
#$strFilter = "(&(objectCategory=User)(|(l=charlotte)(l=dallas))(!department=hr))"
```

24. This concludes this step-by-step exercise.

One Step Further: Controlling How a Script Executes Against Active Directory

In this exercise, we will control the way we return data from Active Directory.

1. To make it easier to keep up the number of users returned from our Active Directory queries, run the DeleteMultipleUsers.ps1 script. This will delete the 60 users we created for the previous step-by-step exercise.

2. Run the Create2000Users.ps1 script. This script will create 2000 users for you to use in the MyTestOU OU.

3. Open the QueryUsersStepbyStep.ps1 script and save it as *yourname*OneStepFurther-QueryUsers.ps1.

4. Because we are not interested in running finely crafted queries in this exercise (rather, we are interested in how to handle large amounts of objects that come back), delete all the *$strFilter* commands except for the one that filters out User objects. This line of code is shown here:

```
$strFilter = "(objectCategory=User)"
```

5. Save and run your script. You will see 1000 user names scroll by in your Windows PowerShell console window. After about 30 seconds (on my machine anyway), you will finally see MyLabUser997 show up. The reason it is MyLabUser997 instead of MyLabUser1000 is that this OU already had three users when we started (myBoss, myDirect1, and myDirect2). This is OK; it is easy to see that the query returned the system default of 1000 objects.

6. We know, however, there are more than 2000 users in the MyTestOU, and we have only been able to retrieve 1000 of them. To get past the query limit that is set for Active Directory, we need to turn on paging. This is simple. We assign a value for the PageSize property to be less than the 1000 object limit. To do this, we use the Item method of the properties collection on the Command object and assign the value of 500 to the PageSize property. This line of code is shown here. Place this code just above this line, which creates the RecordSet object: $objRecordSet = $objCommand.Execute().

```
$objCommand.Properties.item("Page Size") = 500
```

7. After you have made the change, save and run your script. You should see all 2000 user objects show up ... however, the results may be a little jumbled. Without using a *Sort-Object* or specifying the Sort property on the server, the values are not guaranteed to be in order. This script takes about a minute or so on my computer.

8. To tell Active Directory we do not want any size limit, specify the SizeLimit property as 0. We can do this by using the Item method of the properties collection on the Command object. This line of code is shown here:

```
$objCommand.Properties.item("Size Limit") = 0
```

9. To make the script a bit more efficient, change the script to perform an asynchronous query (synchronous being the default). This will reduce the network bandwidth consumed and will even out the processor load on your server. To do this, declare a variable called *$blnTrue* and set it equal to the Boolean type. Assign the value −1 to it. Place this code just under the line that creates the *$strQuery* variable. This line of code is shown here:

```
$blnTrue = [bool]-1
```

10. Under the line of code that sets the size limit, use the Item method of the properties collection to assign the value true to the asynchronous property of the Command object.

Use the Boolean value you created and stored in the *$blnTrue* variable. This line of code is shown here:

```
$objCommand.Properties.item("Asynchronous") = $blnTrue
```

11. Save and run your script. You should see the script run perhaps a little faster because it is doing an asynchronous query. If your script does not run properly, compare your script with the OneStepFurtherQueryUsers.ps1 script.

12. To clean up after this lab, run the Delete2000Users.ps1 script. It will delete the 2000 users we created at the beginning of the exercise.

13. This concludes this one step further exercise.

Chapter 8 Quick Reference

To	Do This
Make an ADO connection into Active Directory	Use the ADsDSOObject provider with ADO to talk to Active Directory
Perform an Active Directory query	Use the Field object to hold attribute data
Tell ADO search to cache results on the client side of the connection	Use the "Cache results" property
Directly query a Global Catalog (GC) server	Use GC:// in your connection moniker, instead of using LDAP://, as shown here: `GC://`
Directly query a specific server in Active Directory	Use LDAP:// in your connection moniker, followed by a trailing backslash (/), as shown here: `LDAP://London/`
Query for multiple attributes in Active Directory using the LDAP dialect	Open a set of parentheses. Inside the set of parentheses, type your attribute name and value for each of the attributes you wish to query. Enclose them in parentheses. At the beginning of the expression between the first two sets of parentheses, use the ampersand (&) operator, as shown here: `(&(objectCategory=computer)(name=london))`
Use server side sorting when using the SQL dialect	Use the order by parameter followed by either the ASC or the DESC keyword, as shown here: `'user'order by sn DESC`
Return more than 1000 objects from an Active Directory ADO query	Turn on paging by specifying the PageSize property on the Command object, and supply a value for SizeLimit property
Connect to Active Directory using alternative credentials	Specify the User ID and Password properties on the Connection object