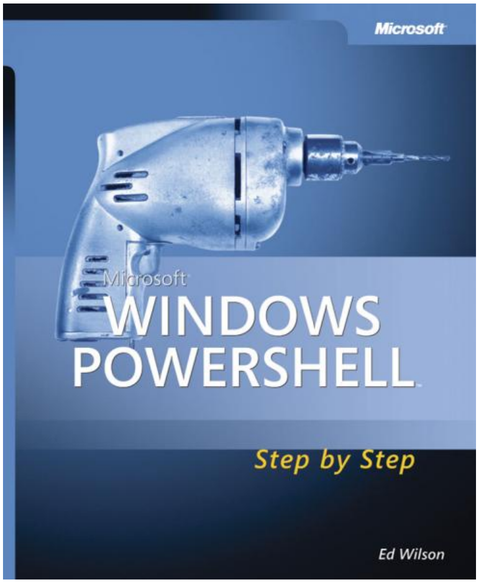


Microsoft® Windows PowerShell™ Step By Step

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

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.

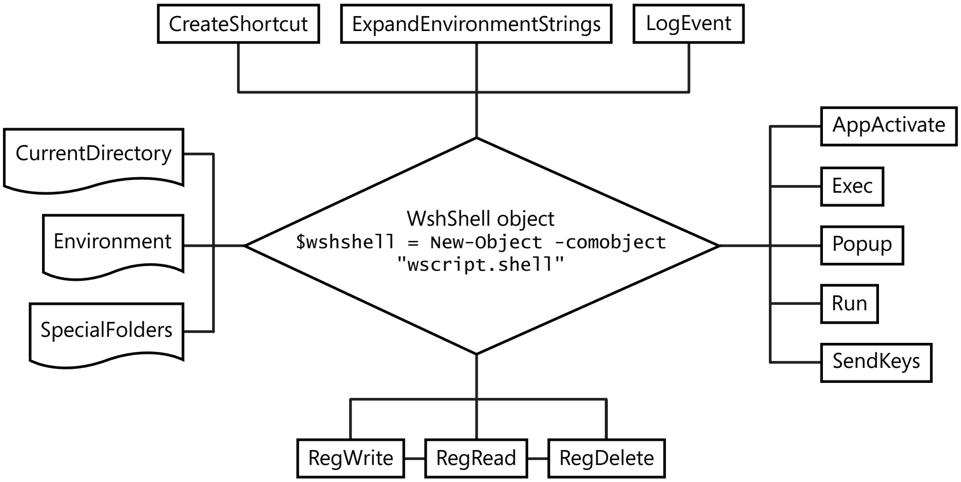


Figure 2-1 The VBScript wshShell object contributes many easy-to-use methods and properties for the network administrator

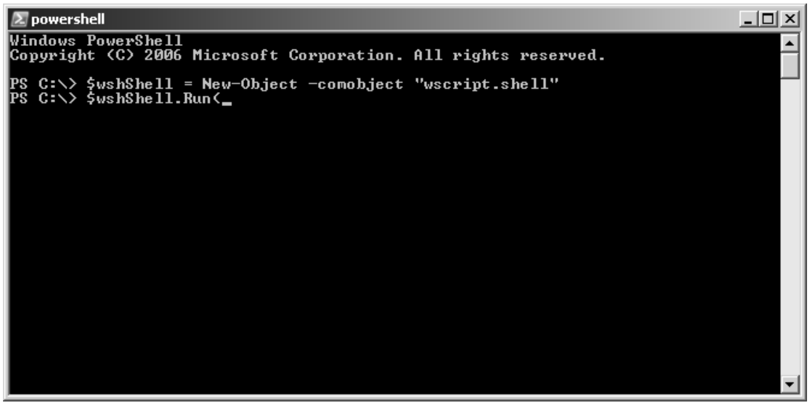
Just the Steps To create a new instance of the wshShell object, use the *New-Object* cmdlet while specifying the -comobject argument and supplying the program ID of "wscript.shell". Hold the object created in a variable. Example:

```
$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.



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

PS C:\> $wshShell = New-Object -comobject "wscript.shell"
PS C:\> $wshShell.Run(_
```

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:

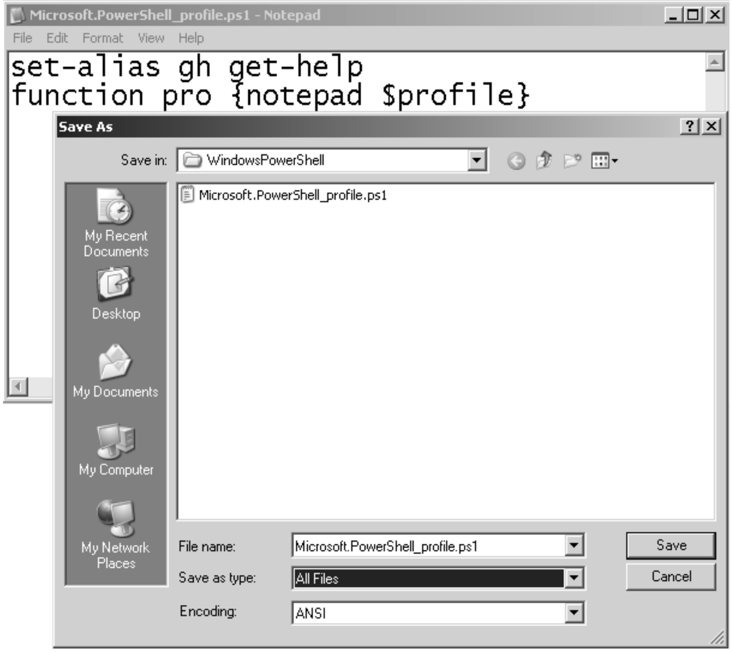
   A useful alias such as *gh* for *Get-Help*. This is shown here:

   ```
   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.



Microsoft.PowerShell_profile.ps1 - Notepad

```
set-alias gh get-help
function pro {notepad $profile}
```

Save As

Save in: WindowsPowerShell

Microsoft.PowerShell_profile.ps1

File name: Microsoft.PowerShell_profile.ps1

Save as type: All Files

Encoding: ANSI

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 UserDomain 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 UserDomain 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 subfolders	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