1

Getting Started with Windows PowerShell

COPYRIGHTED MATERIAL

If you are like me, then when you begin to look seriously at an interesting piece of software, you like to get your hands dirty and play with it from the beginning. In this chapter, I show you how to get started using Windows PowerShell, and I'll show you enough of the PowerShell commands to let you begin to find your way around effectively. In the rest of the book, I help you build on that initial knowledge so that you can use PowerShell for a wide range of useful tasks, depending on your requirements.

Windows PowerShell, as you probably already know, is Microsoft's new command shell and scripting language. It provides a command line environment for interactive exploration and administration of computers, and by storing and running Windows PowerShell commands in a script file, you can run scripts to carry out administrative tasks multiple times. Windows PowerShell differs in detail from existing command line environments on the Windows and Unix platforms, although it has similarities to past environments. In Chapter 3, in particular, I explain more about the PowerShell approach, although differences from existing command shells and scripting languages will emerge in every chapter.

Once you have had a brief taste of PowerShell, you will need to understand a little of the assumptions and approach that lie behind the design decisions that have made PowerShell the useful tool that it is. In Chapter 2, I step back from using the PowerShell command line and look at the strengths and deficiencies of some existing Microsoft approaches to system management and then, in Chapter 3, take a look at the philosophy and practical thought that lies behind the approach taken in Windows PowerShell.

Installing Windows PowerShell

Windows PowerShell depends on the presence of the .NET Framework 2.0. Before you install PowerShell, you need to be sure that you have the .NET Framework 2.0 installed.

Installing .NET Framework 2.0

At the time of writing, the 32-bit version of the .NET Framework 2.0 runtime is available for downloading from `www.microsoft.com/downloads/details.aspx?FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en`.

If you are using 64-bit Itanium processors, download the .NET Framework 2.0 runtime from `www.microsoft.com/downloads/details.aspx?familyid=53C2548B-BEC7-4AB4-8CBE-33E07CFC83A7&displaylang=en`. Windows PowerShell is only available on Windows Server 2003 for Itanium processors.

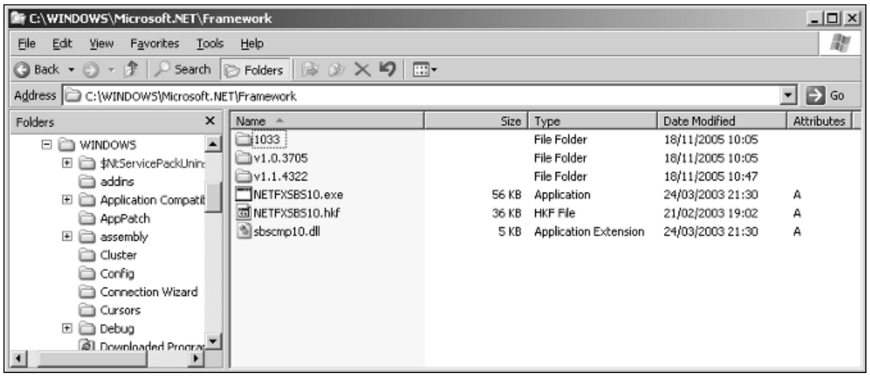
If you are using AMD 64-bit processors, download the runtime from `www.microsoft.com/downloads/info.aspx?na=47&p=3&SrcDisplayLang=en&SrcCategoryId=&SrcFamilyId=F4DD601B-1B88-47A3-BDC1-79AFA79F6FB0&u=details.aspx%3ffamilyid%3dB44A0000-ACF8-4FA1-AFFB-40E78D0788B00%26displaylang%3den`.

If you are unsure whether or not you have .NET Framework 2.0 installed, navigate to `C:\Windows\Microsoft.NET\Framework` *(if necessary substitute another drive letter if your system drive is not drive C:). In that folder you will find folders that contain the versions of the .NET Framework that are installed on your machine. If you see a folder named v2.0.50727, then you have the .NET Framework 2.0 installed. The .NET Framework 2.0 SDK, which you can download separately, is useful as a source of information on .NET 2.0 classes that you can use with PowerShell.*

If you want to install the 32 bit .NET Framework 2.0 Software Development Kit (SDK), download it from `www.microsoft.com/downloads/details.aspx?FamilyID=fe6f2099-b7b4-4f47-a244-c96d69c35dec&displaylang=en`**. To install the .NET Framework 2.0 SDK, you must first install the 32-bit runtime.**

There are also 64-bit versions of the .NET Framework 2.0 SDK available for downloading. The version of the runtime for Itanium is located at `www.microsoft.com/downloads/details.aspx?familyid=F4DD601B-1B88-47A3-BDC1-79AFA79F6FB0&displaylang=en`**. The 64-bit version for AMD processors is located at** `www.microsoft.com/downloads/details.aspx?familyid=1AEF6FCE-6E06-4B66-AFE4-9AAD3C835D3D&displaylang=en`**.**

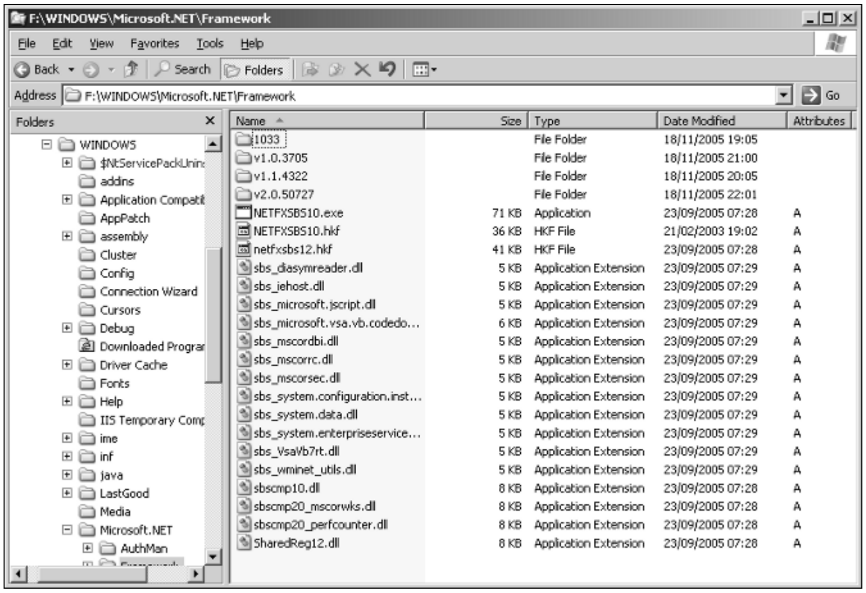
Figure 1-1 shows what you would expect to see in the Framework folder on a clean install of Windows 2003 Service Pack 1 which does not have the .NET Framework 2.0 runtime installed.



Name	Size	Type	Date Modified	Attributes
1033		File Folder	18/11/2005 10:05	
v1.0.3705		File Folder	18/11/2005 10:05	
v1.1.4322		File Folder	18/11/2005 10:47	
NETFXSBS10.exe	56 KB	Application	24/03/2003 21:30	A
NETFXSBS10.hkf	36 KB	HKF File	21/02/2003 19:02	A
sbscmp10.dll	5 KB	Application Extension	24/03/2003 21:30	A

Figure 1-1

Figure 1-2 shows the appearance of the Framework folder on a clean install of Windows 2003 Service Pack 1 after the .NET Framework 2.0 runtime has been installed.



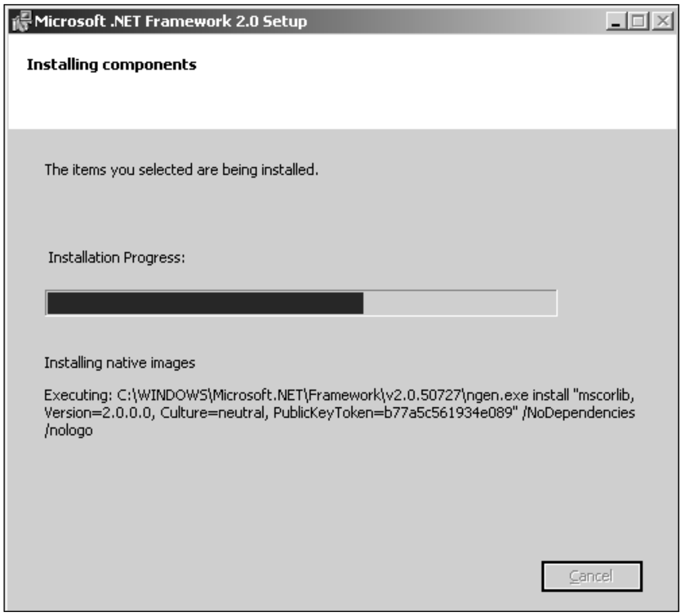
Name	Size	Type	Date Modified	Attributes
1033		File Folder	18/11/2005 19:05	
v1.0.3705		File Folder	18/11/2005 21:00	
v1.1.4322		File Folder	18/11/2005 20:05	
v2.0.50727		File Folder	18/11/2005 22:01	
NETFXSBS10.exe	71 KB	Application	23/09/2005 07:28	A
NETFXSBS10.hkf	36 KB	HKF File	21/02/2003 19:02	A
netfxsbs12.hkf	41 KB	HKF File	23/09/2005 07:28	A
sbs_diasymreader.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbs_iehost.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbs_microsoft.jscript.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbs_microsoft.vsa.vb.codedo...	6 KB	Application Extension	23/09/2005 07:29	A
sbs_mscordbi.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbs_mscorrc.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbs_mscorsec.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbs_system.configuration.inst...	5 KB	Application Extension	23/09/2005 07:29	A
sbs_system.data.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbs_system.enterpriseservice...	5 KB	Application Extension	23/09/2005 07:29	A
sbs_VsaVb7rt.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbs_wminet_utils.dll	5 KB	Application Extension	23/09/2005 07:29	A
sbscmp10.dll	8 KB	Application Extension	23/09/2005 07:28	A
sbscmp20_mscorwks.dll	8 KB	Application Extension	23/09/2005 07:28	A
sbscmp20_perfcounter.dll	8 KB	Application Extension	23/09/2005 07:28	A
SharedReg12.dll	8 KB	Application Extension	23/09/2005 07:28	A

Figure 1-2

You don't need to delete the v1.0.3705 or v1.1.4322 folders. In fact, you are likely to cause problems for applications that need earlier versions of the .NET Framework if you delete those folders. The .NET Framework 2.0 is designed to run side by side with .NET Framework 1.0 and 1.1.

To install the .NET Framework 2.0, follow these steps.

1. Navigate in Windows Explorer to the folder where you downloaded the installer, `dotnetfx.exe`.
2. Double-click the installer. On the splash screen that opens, click Next.
3. On the EULA screen, accept the license agreement and click Install.
4. The installer then proceeds to install the .NET Framework 2.0, as shown in Figure 1-3.
5. When the installation has completed successfully, you should see a screen similar to Figure 1-4.
6. If you have Internet connectivity, click the Product Support Center link shown in Figure 1-4 to check for any updates.

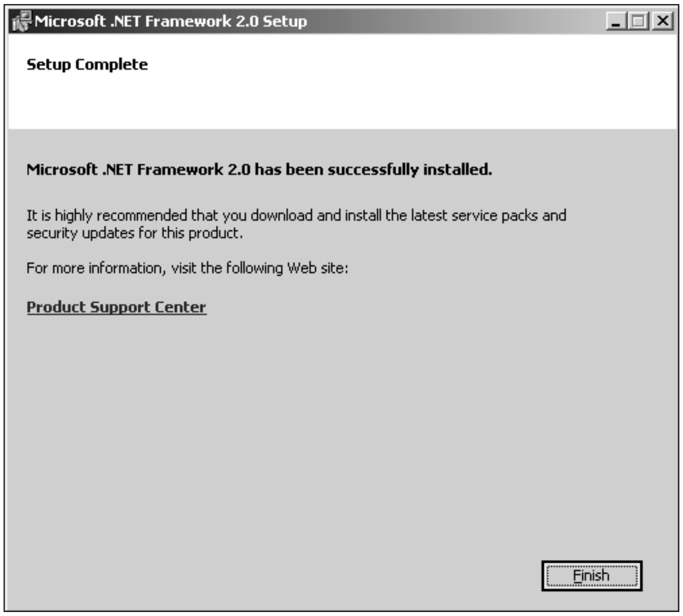


The items you selected are being installed.

Installing native images

Executing: C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ngen.exe install "mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" /NoDependencies /nologo

Figure 1-3



Microsoft .NET Framework 2.0 has been successfully installed.

It is highly recommended that you download and install the latest service packs and security updates for this product.

For more information, visit the following Web site:

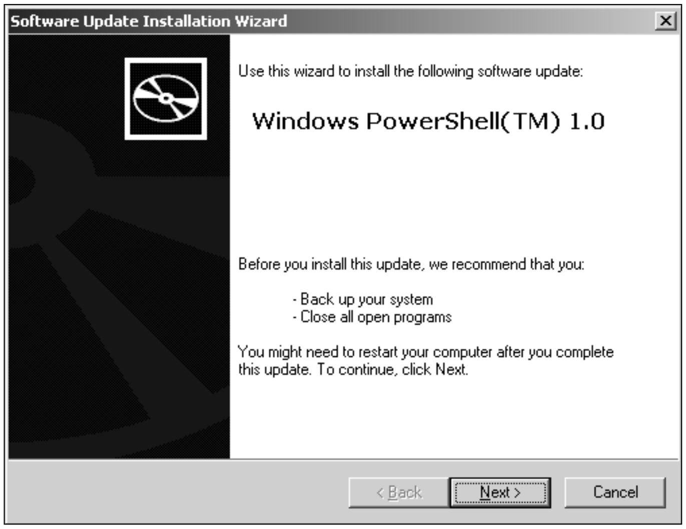
Figure 1-4

Once you have installed the .NET Framework 2.0, you can then install Windows PowerShell.

Installing Windows PowerShell

To install Windows PowerShell on a 32-bit system, follow these steps. If you are installing it on a 64-bit system, the installer filename will differ.

1. Double-click the `.exe` installer file appropriate for the version of Windows PowerShell you want to install. The initial screen of the installation wizard, similar to the one shown in Figure 1-5, is displayed.



Software Update Installation Wizard

Use this wizard to install the following software update:

Windows PowerShell(TM) 1.0

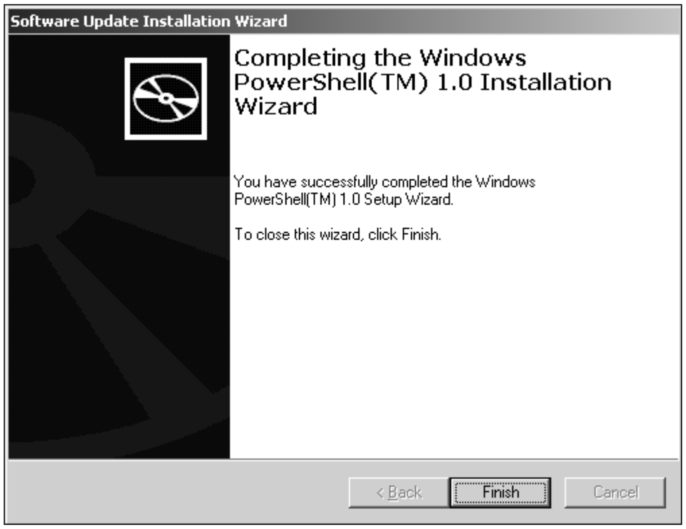
Before you install this update, we recommend that you:

- Back up your system
- Close all open programs

You might need to restart your computer after you complete this update. To continue, click Next.

Figure 1-5

2. Click Next.
3. Accept the license agreement and click Next.
4. If you are installing on drive C: on a 32-bit system, the default install location is `C:\Windows\System32\windowspowershell\v1.0.`
5. When the installation has completed successfully you will see a screen similar to Figure 1-6.
6. Click Finish.



Completing the Windows PowerShell(TM) 1.0 Installation Wizard

You have successfully completed the Windows PowerShell(TM) 1.0 Setup Wizard.

To close this wizard, click Finish.

Figure 1-6

Starting and Stopping PowerShell

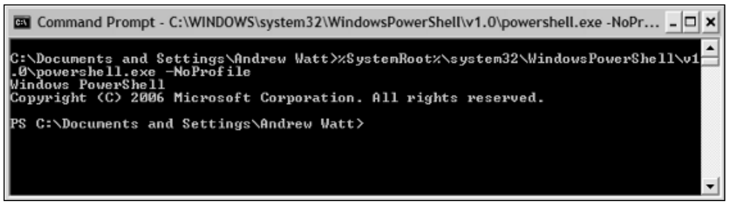
Once you have installed Windows PowerShell, you have several options for starting it.

Starting PowerShell

To start PowerShell without using any profile file to customize its behavior, open a command window (On Windows 2003, select Start ➪ All Programs ➪ Accessories ➪ Command Prompt), then type:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\powershell.exe -NoProfile
```

After a short pause, the Windows PowerShell prompt should appear (see Figure 1-7).

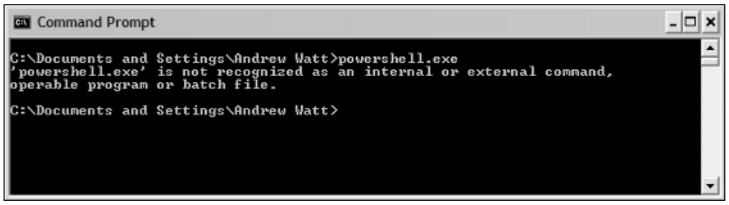

Command Prompt - C:\WINDOWS\system32\WindowsPowerShell\v1.0\powershell.exe -NoPr...

```
C:\Documents and Settings\Andrew Watt>%SystemRoot%\system32\WindowsPowerShell\v1
.0\powershell.exe -NoProfile
Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.

PS C:\Documents and Settings\Andrew Watt>
```

Figure 1-7

If you are still using a Release Candidate and attempt to start PowerShell by simply typing `PowerShell .exe` at the command shell prompt, you may see the error message shown in Figure 1-8. To fix that, update to the final release version.

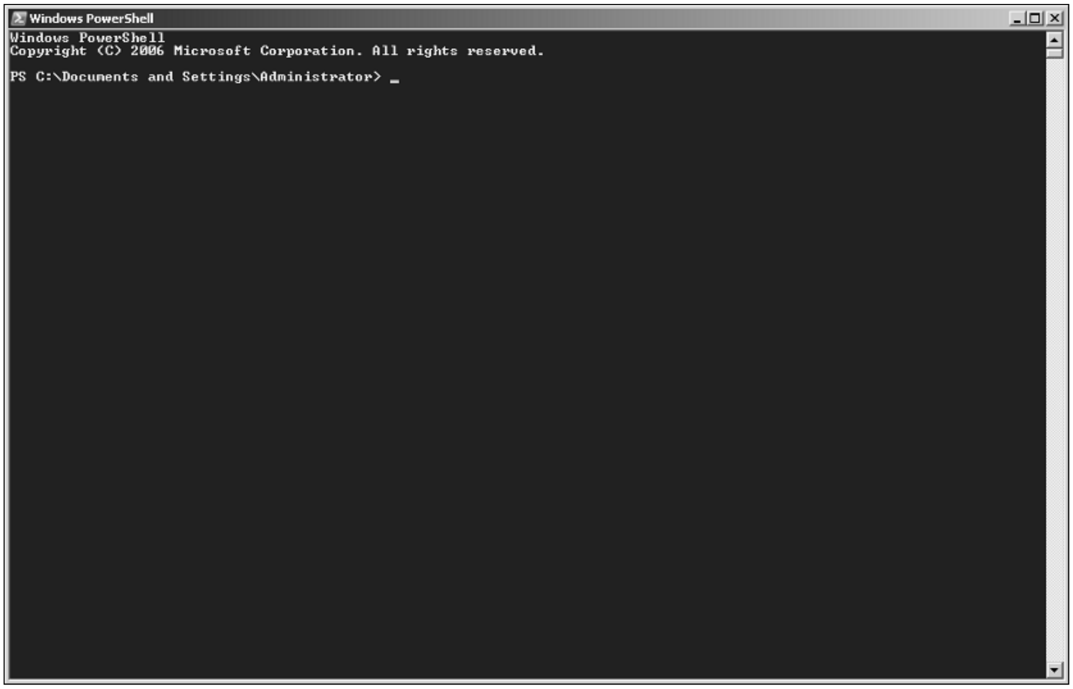


```
C:\Documents and Settings\Andrew Watt>powershell.exe
'powershell.exe' is not recognized as an internal or external command,
operable program or batch file.

C:\Documents and Settings\Andrew Watt>
```

Figure 1-8

Alternatively, you can start PowerShell by selecting Start ➪ All Programs ➪ Windows PowerShell 1.0 ➪ Windows PowerShell (see Figure 1-9).



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

PS C:\Documents and Settings\Administrator> _
```

Figure 1-9

Because of security concerns about previous Microsoft scripting technologies, the default setting of Windows PowerShell is that scripting is locked down. Specifically, when Windows PowerShell starts, it does not attempt to run profile files (which are PowerShell scripts) that contain various settings controlling how PowerShell should run. Whichever way you start PowerShell initially, you will probably later want to enable scripts. To do that, you use the `set-executionpolicy` cmdlet. Type:

```
set-executionpolicy -ExecutionPolicy "RemoteSigned"
```

and you will be able to run locally created scripts without signing them. I cover execution policy in more detail in Chapter 10.

There are several additional options for starting PowerShell, and I will briefly describe all of those — after I show you how to stop PowerShell.

Exiting PowerShell

To stop PowerShell, simply type the following at the PowerShell command line:

```
Exit
```

and you are returned to the `CMD.exe` command prompt (assuming that you started PowerShell from the `CMD.exe` prompt). If you started PowerShell using Start ➪ All Programs ➪ Windows PowerShell 1.0 ➪ Windows PowerShell, the PowerShell window closes.

You can't use "`quit`" to exit PowerShell. It just causes an error message to be displayed.

Startup Options

You have several options for how you start PowerShell. These are listed and explained in the following table. On the command line, each parameter name is preceded by a minus sign.

Parameter	Explanation
`Command`	The value of the `Command` parameter is to be executed as if it were typed at a PowerShell command prompt.
`Help`	Displays information about the startup options for PowerShell summarized in this table.
`InputFormat`	Specifies the format of any input data. The options are "Text" and "XML."
`NoExit`	Specifies that PowerShell doesn't exit after the command you enter has been executed. Specify the `NoExit` parameter before the `Command` parameter.
`NoLogo`	The copyright message usually displayed when PowerShell starts is omitted. Specifying this parameter causes the copyright message not to be displayed.
`NonInteractive`	Use this parameter when no user input is needed nor any output to the console.
`NoProfile`	The user initialization scripts are not run.
`OutputFormat`	Specifies the format for outputting data. The options are "Text" and "XML."
`PSConsoleFile`	Specifies a Windows PowerShell console file to run at startup.

To view information about all help options, type:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\powershell.exe -Help
```

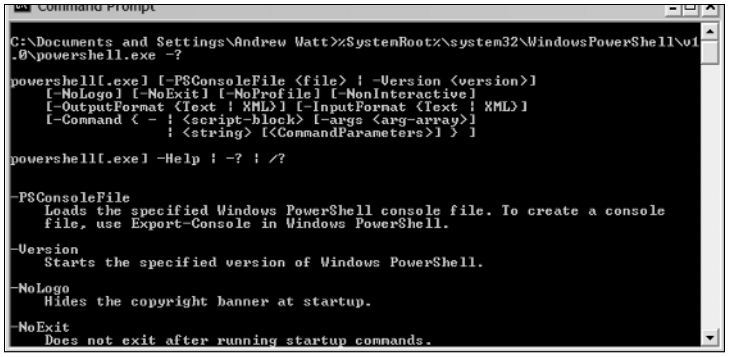
or:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\powershell.exe -?
```

or:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\powershell.exe /?
```

at the command line. Each of those commands will cause the information, part of which is shown in Figure 1-10, to be displayed.



```
C:\Documents and Settings\Andrew Watt>%SystemRoot%\system32\WindowsPowerShell\v1
.0\powershell.exe -?

powershell[.exe] [-PSConsoleFile <file> | -Version <version>]
    [-NoLogo] [-NoExit] [-NoProfile] [-NonInteractive]
    [-OutputFormat {Text | XML}] [-InputFormat {Text | XML}]
    [-Command { - | <script-block> [-args <arg-array>]
                  | <string> [<CommandParameters>] } ]

powershell[.exe] -Help | -? | /?


-PSConsoleFile
    Loads the specified Windows PowerShell console file. To create a console
    file, use Export-Console in Windows PowerShell.

-Version
    Starts the specified version of Windows PowerShell.

-NoLogo
    Hides the copyright banner at startup.

-NoExit
    Does not exit after running startup commands.
```

Figure 1-10

Notice that there are sets of parameters that you can use with `PowerShell.exe`. You can combine parameters only in the ways shown in Figure 1-10.

The preceding commands give you help on how to start Windows PowerShell. Once you start PowerShell, you also need to know where to find help on individual PowerShell commands. As a first step, you need to be able to find out what commands are available to you.

Individual PowerShell commands are small and granular. As a result, they are called cmdlets (pronounced "commandlets").

Finding Available Commands

In this section, I will show you a few commonly used commands and show you how to explore the PowerShell cmdlets to see what PowerShell commands are available on your system. The `get-command` cmdlet allows you to explore the commands available to you in Windows PowerShell.

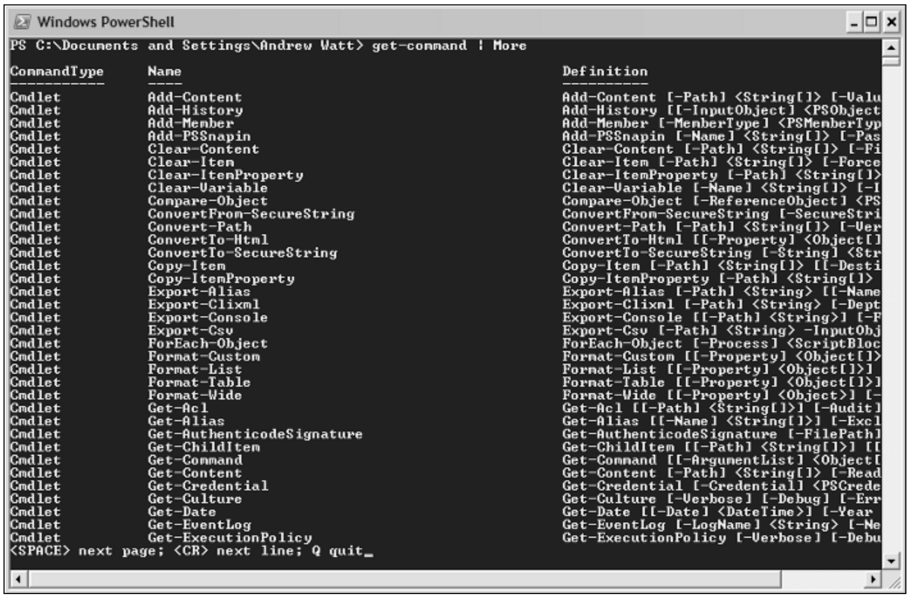
The simplest, but not the most useful, way to use the `get-command` cmdlet is simply to type:

```
get-command
```

at the PowerShell command line. Several screens of command names scroll past when you do this — there are a lot of cmdlets in PowerShell. It's more useful to view the information one screen at a time. You achieve that by typing:

```
get-command | More
```

at the PowerShell command line. The result is similar to that shown in Figure 1-11. If you run that command and carefully read the available commands, you will get some idea of the scope of functionality that PowerShell allows you to control and manage.



```
PS C:\Documents and Settings\Andrew Watt> get-command | More

CommandType     Name                                           Definition
-----------     ----                                           ----------
Cmdlet          Add-Content                                    Add-Content [-Path] <String[]> [-Valu
Cmdlet          Add-History                                    Add-History [[-InputObject] <PSObject
Cmdlet          Add-Member                                     Add-Member [-MemberType] <PSMemberTyp
Cmdlet          Add-PSSnapin                                   Add-PSSnapin [-Name] <String[]> [-Pas
Cmdlet          Clear-Content                                  Clear-Content [-Path] <String[]> [-Fi
Cmdlet          Clear-Item                                     Clear-Item [-Path] <String[]> [-Force
Cmdlet          Clear-ItemProperty                             Clear-ItemProperty [-Path] <String[]>
Cmdlet          Clear-Variable                                 Clear-Variable [-Name] <String[]> [-I
Cmdlet          Compare-Object                                 Compare-Object [-ReferenceObject] <PS
Cmdlet          ConvertFrom-SecureString                       ConvertFrom-SecureString [-SecureStri
Cmdlet          Convert-Path                                   Convert-Path [-Path] <String[]> [-Ver
Cmdlet          ConvertTo-Html                                 ConvertTo-Html [[-Property] <Object[]
Cmdlet          ConvertTo-SecureString                         ConvertTo-SecureString [-String] <Str
Cmdlet          Copy-Item                                      Copy-Item [-Path] <String[]> [[-Desti
Cmdlet          Copy-ItemProperty                              Copy-ItemProperty [-Path] <String[]>
Cmdlet          Export-Alias                                   Export-Alias [-Path] <String> [[-Name
Cmdlet          Export-Clixml                                  Export-Clixml [-Path] <String> [-Dept
Cmdlet          Export-Console                                 Export-Console [[-Path] <String>] [-F
Cmdlet          Export-Csv                                     Export-Csv [-Path] <String> -InputObj
Cmdlet          ForEach-Object                                 ForEach-Object [-Process] <ScriptBloc
Cmdlet          Format-Custom                                  Format-Custom [[-Property] <Object[]>
Cmdlet          Format-List                                    Format-List [[-Property] <Object[]>]
Cmdlet          Format-Table                                   Format-Table [[-Property] <Object[]>]
Cmdlet          Format-Wide                                    Format-Wide [[-Property] <Object>] [-
Cmdlet          Get-Acl                                        Get-Acl [[-Path] <String[]>] [-Audit]
Cmdlet          Get-Alias                                      Get-Alias [[-Name] <String[]>] [-Excl
Cmdlet          Get-AuthenticodeSignature                      Get-AuthenticodeSignature [-FilePath]
Cmdlet          Get-ChildItem                                  Get-ChildItem [[-Path] <String[]>] [[
Cmdlet          Get-Command                                    Get-Command [[-ArgumentList] <Object[
Cmdlet          Get-Content                                    Get-Content [-Path] <String[]> [-Read
Cmdlet          Get-Credential                                 Get-Credential [-Credential] <PSCrede
Cmdlet          Get-Culture                                    Get-Culture [-Verbose] [-Debug] [-Err
Cmdlet          Get-Date                                       Get-Date [[-Date] <DateTime>] [-Year
Cmdlet          Get-EventLog                                   Get-EventLog [-LogName] <String> [-Ne
Cmdlet          Get-ExecutionPolicy                            Get-ExecutionPolicy [-Verbose] [-Debu
<SPACE> next page; <CR> next line; Q quit_
```

Figure 1-11

To view another screen of commands, press the spacebar once. Repeat this to view each additional screen of commands.

PowerShell commands are formed of a verb, followed by a hyphen (or minus sign), followed by a noun. The `get-command` *cmdlet illustrates the structure. The verb "*`get`*" is followed by a hyphen, which is followed by a noun "*`command`*." PowerShell uses the singular form of the noun, even when, as is often the case, you want to find multiple items that satisfy your requirements. Thus, you might use* `get-process` *to get all the processes running on a system, as opposed to* `get-processes`*.*

You can use wildcards to focus your search for the relevant command. For example, to find all commands that use the `get` verb, use the following command:

```
get-command get-*
```

or, the slightly tidier:

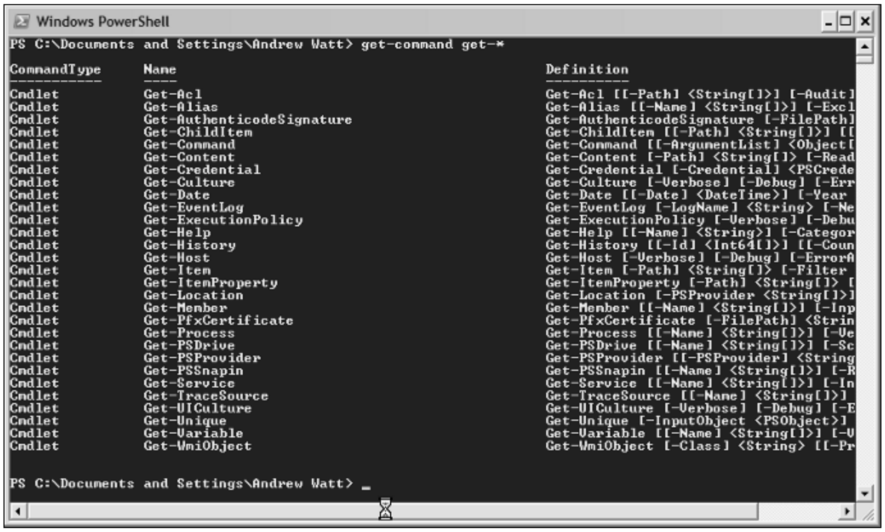
```
get-command get-* | More
```

The argument to the `get-command` cmdlet uses the `*` wildcard. The argument `get-*` finds any command whose name begins with get, a hyphen, and zero or more other characters. As you can see in Figure 1-12, there are many cmdlets that use the `get` verb.

Other verbs worth looking for include `add`, `format`, `new`, `set`, and `write`. To see a complete list of available verbs, type the following command:

```
get-command | group-object verb
```

Figure 1-13 shows the results. The preceding command uses a pipeline that consists of two steps. The first uses the `get-command` cmdlet to create objects representing all available commands. The second step uses the `group-object` cmdlet to group the results by the verb.

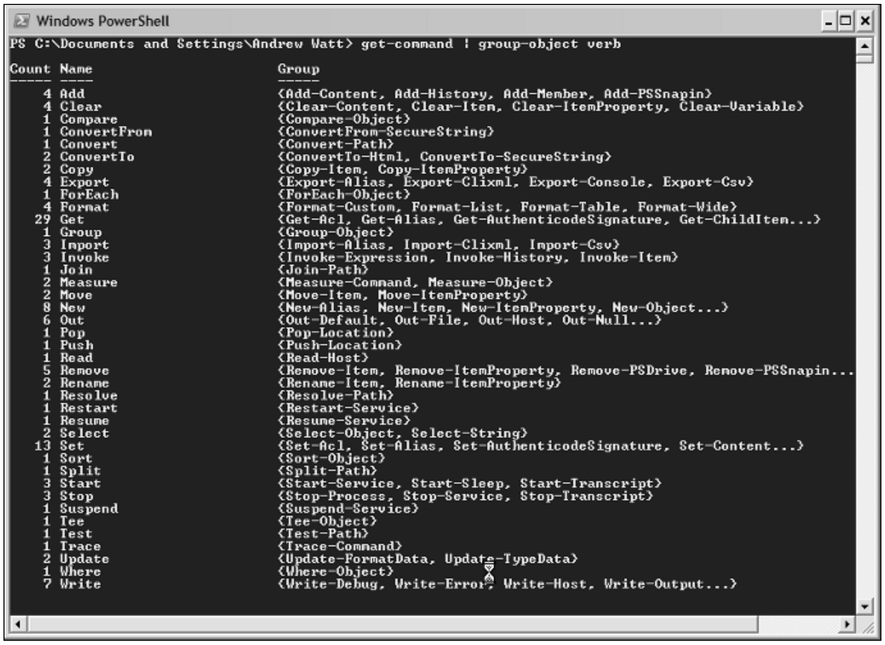


```
PS C:\Documents and Settings\Andrew Watt> get-command get-*

CommandType     Name                                             Definition
-----------     ----                                             ----------
Cmdlet          Get-Acl                                          Get-Acl [[-Path] <String[]>] [-Audit]
Cmdlet          Get-Alias                                        Get-Alias [[-Name] <String[]>] [-Excl
Cmdlet          Get-AuthenticodeSignature                        Get-AuthenticodeSignature [-FilePath]
Cmdlet          Get-ChildItem                                    Get-ChildItem [[-Path] <String[]>] [[
Cmdlet          Get-Command                                      Get-Command [[-ArgumentList] <Object[
Cmdlet          Get-Content                                      Get-Content [-Path] <String[]> [-Read
Cmdlet          Get-Credential                                   Get-Credential [-Credential] <PSCrede
Cmdlet          Get-Culture                                      Get-Culture [-Verbose] [-Debug] [-Err
Cmdlet          Get-Date                                         Get-Date [[-Date] <DateTime>] [-Year
Cmdlet          Get-EventLog                                     Get-EventLog [-LogName] <String> [-Ne
Cmdlet          Get-ExecutionPolicy                              Get-ExecutionPolicy [-Verbose] [-Debu
Cmdlet          Get-Help                                         Get-Help [[-Name] <String>] [-Categor
Cmdlet          Get-History                                      Get-History [[-Id] <Int64[]>] [[-Coun
Cmdlet          Get-Host                                         Get-Host [-Verbose] [-Debug] [-ErrorA
Cmdlet          Get-Item                                         Get-Item [-Path] <String[]> [-Filter
Cmdlet          Get-ItemProperty                                 Get-ItemProperty [-Path] <String[]> [
Cmdlet          Get-Location                                     Get-Location [-PSProvider <String[]>]
Cmdlet          Get-Member                                       Get-Member [[-Name] <String[]>] [-Inp
Cmdlet          Get-PfxCertificate                               Get-PfxCertificate [-FilePath] <Strin
Cmdlet          Get-Process                                      Get-Process [[-Name] <String[]>] [-Ve
Cmdlet          Get-PSDrive                                      Get-PSDrive [[-Name] <String[]>] [-Sc
Cmdlet          Get-PSProvider                                   Get-PSProvider [[-PSProvider] <String
Cmdlet          Get-PSSnapin                                     Get-PSSnapin [[-Name] <String[]>] [-R
Cmdlet          Get-Service                                      Get-Service [[-Name] <String[]>] [-In
Cmdlet          Get-TraceSource                                  Get-TraceSource [[-Name] <String[]>]
Cmdlet          Get-UICulture                                    Get-UICulture [-Verbose] [-Debug] [-E
Cmdlet          Get-Unique                                       Get-Unique [-InputObject <PSObject>]
Cmdlet          Get-Variable                                     Get-Variable [[-Name] <String[]>] [-V
Cmdlet          Get-WmiObject                                    Get-WmiObject [-Class] <String> [[-Pr


PS C:\Documents and Settings\Andrew Watt> _
```

Figure 1-12



```
PS C:\Documents and Settings\Andrew Watt> get-command | group-object verb

Count Name                      Group
----- ----                      -----
    4 Add                       {Add-Content, Add-History, Add-Member, Add-PSSnapin}
    4 Clear                     {Clear-Content, Clear-Item, Clear-ItemProperty, Clear-Variable}
    1 Compare                   {Compare-Object}
    1 ConvertFrom               {ConvertFrom-SecureString}
    1 Convert                   {Convert-Path}
    2 ConvertTo                 {ConvertTo-Html, ConvertTo-SecureString}
    2 Copy                      {Copy-Item, Copy-ItemProperty}
    4 Export                    {Export-Alias, Export-Clixml, Export-Console, Export-Csv}
    1 ForEach                   {ForEach-Object}
    4 Format                    {Format-Custom, Format-List, Format-Table, Format-Wide}
   29 Get                       {Get-Acl, Get-Alias, Get-AuthenticodeSignature, Get-ChildItem...}
    1 Group                     {Group-Object}
    3 Import                    {Import-Alias, Import-Clixml, Import-Csv}
    3 Invoke                    {Invoke-Expression, Invoke-History, Invoke-Item}
    1 Join                      {Join-Path}
    2 Measure                   {Measure-Command, Measure-Object}
    2 Move                      {Move-Item, Move-ItemProperty}
    8 New                       {New-Alias, New-Item, New-ItemProperty, New-Object...}
    6 Out                       {Out-Default, Out-File, Out-Host, Out-Null...}
    1 Pop                       {Pop-Location}
    1 Push                      {Push-Location}
    1 Read                      {Read-Host}
    5 Remove                    {Remove-Item, Remove-ItemProperty, Remove-PSDrive, Remove-PSSnapin...
    2 Rename                    {Rename-Item, Rename-ItemProperty}
    1 Resolve                   {Resolve-Path}
    1 Restart                   {Restart-Service}
    1 Resume                    {Resume-Service}
    2 Select                    {Select-Object, Select-String}
   13 Set                       {Set-Acl, Set-Alias, Set-AuthenticodeSignature, Set-Content...}
    1 Sort                      {Sort-Object}
    1 Split                     {Split-Path}
    3 Start                     {Start-Service, Start-Sleep, Start-Transcript}
    3 Stop                      {Stop-Process, Stop-Service, Stop-Transcript}
    1 Suspend                   {Suspend-Service}
    1 Tee                       {Tee-Object}
    1 Test                      {Test-Path}
    1 Trace                     {Trace-Command}
    2 Update                    {Update-FormatData, Update-TypeData}
    1 Where                     {Where-Object}
    7 Write                     {Write-Debug, Write-Error, Write-Host, Write-Output...}
```

Figure 1-13

To find the nouns available in your installation of PowerShell, use the following command:

```
get-command | group-object noun
```

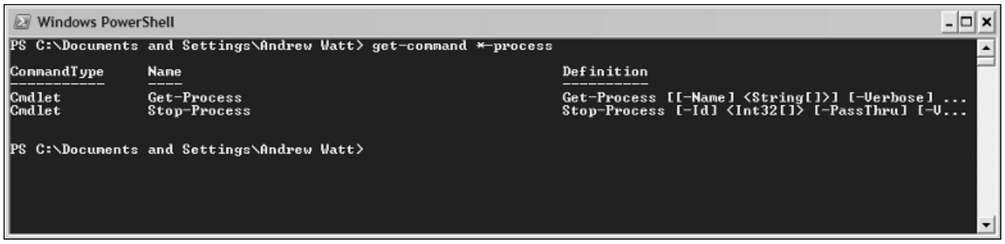
If you want to sort the nouns alphabetically use the following command:

```
get-command | group-object noun | sort-object name
```

You can also use the `get-command` cmdlet to explore in other ways. For example, suppose that you want to find all cmdlets that you can use to work with processes. The preceding command shows you that `process` is one of the nouns used in cmdlets. One way to display information about all cmdlets that operate on processes is to use the following command:

```
get-command *-process
```

Figure 1-14 shows you that there are only two cmdlets that you can use to work specifically with processes. As you construct pipelines with multiple steps, you have many other cmdlets available for use with process-related information.



```
PS C:\Documents and Settings\Andrew Watt> get-command *-process

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

PS C:\Documents and Settings\Andrew Watt>
```

Figure 1-14

You can adapt the preceding command to find cmdlets relevant to other nouns. For example, the command:

```
get-command *-service
```

will find all cmdlets that relate to services.

Getting Help

When you're using PowerShell, you need to be able to find out how to use commands that you are already aware of or that you find by using the techniques described in the previous section.

You use the `get-help` cmdlet to get help information about individual cmdlets. You can use the `get-help` cmdlet with or without parameters. Using the `get-help` cmdlet with no parameters displays abbreviated help information.

For example, to get help on the `get-process` cmdlet type either:

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

or:

```
get-process -?
```

at the PowerShell command line.

The default behavior of the `get-help` cmdlet when providing help information about a specific command is to dump all the help text to the screen at once, causing anything that won't fit on one screen to scroll off the screen and out of sight. You may find it more useful to display the help information one screen at a time by using `More`:

```
    get-help get-process | More
```

or:

```
    get-process -? | More
```

You are likely to have the `help` function available to you. It behaves similarly to the `get-help` cmdlet, except that the `help` function displays the help information one screen at a time. To display the help information for the `get-process` cmdlet one screen at a time, you can type:

```
    help get-process
```

Since that is a little shorter to type than the `get-help` syntax, you may find that it's more convenient.

PowerShell displays help information in a way similar to `man` in Unix. The help for each command or other piece of syntax is structured in the following sections:

- **Name** – The name of the cmdlet
- **Synopsis** – A brief text description of the cmdlet
- **Syntax** – Demonstrates how the cmdlet can be used
- **Detailed Description** – A longer text description of the cmdlet
- **Parameters** – Provides detailed information about how to use each parameter
- **Input Type** – Specifies the type of the input object(s)
- **Return Type** – Specifies the type of the returned object
- **Examples** – Examples of how to use the cmdlet
- **Related Links** – Names of other cmdlets with related functionality
- **Remarks** – Information about using parameters with the cmdlet

For some commands, some sections may contain no help information.

When you use no parameter with the `get-help` cmdlet, you see the following sections of information:

- ❑ Name
- ❑ Synopsis
- ❑ Syntax
- ❑ Detailed Description
- ❑ Related Links
- ❑ Remarks

If you use the `-detailed` parameter, for example:

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

you see the following sections of help information:

- ❑ Name
- ❑ Synopsis
- ❑ Syntax
- ❑ Detailed Description
- ❑ Parameters
- ❑ Examples
- ❑ Remarks

If you use the `-full` parameter, for example:

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

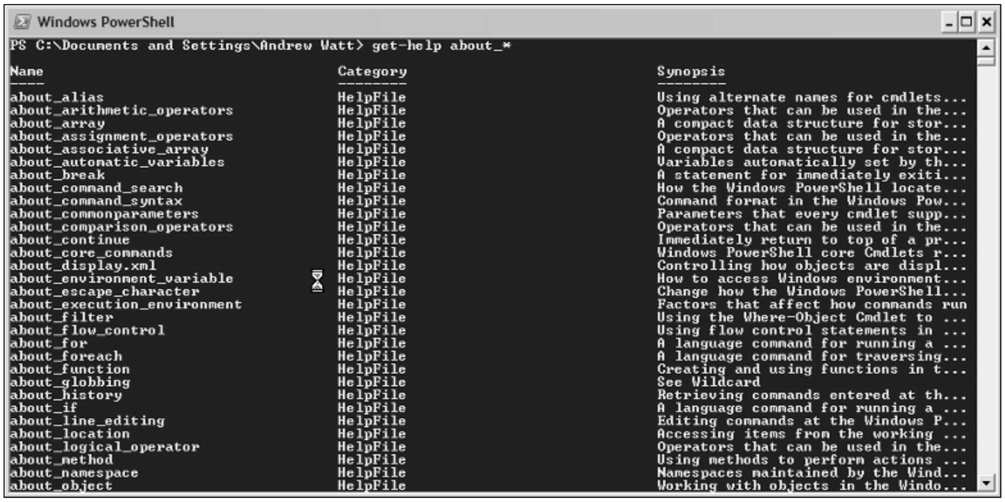
you see the following sections of help information:

- ❑ Name
- ❑ Synopsis
- ❑ Syntax
- ❑ Detailed Description
- ❑ Parameters
- ❑ Input Type
- ❑ Return Type
- ❑ Notes
- ❑ Examples
- ❑ Related Links

In addition to the built-in help about cmdlets, you can also access help about aspects of the PowerShell scripting language using the `get-help` cmdlet. If you don't know what help files on the language are available, use the command:

```
get-help about_* | more
```

to display them. Figure 1-15 shows one screen of results. This works, since each of these help files begins with `about_`.



```
PS C:\Documents and Settings\Andrew Watt> get-help about_*

Name                              Category                          Synopsis
----                              --------                          --------
about_alias                       HelpFile                          Using alternate names for cmdlets...
about_arithmetic_operators        HelpFile                          Operators that can be used in the...
about_array                       HelpFile                          A compact data structure for stor...
about_assignment_operators        HelpFile                          Operators that can be used in the...
about_associative_array           HelpFile                          A compact data structure for stor...
about_automatic_variables         HelpFile                          Variables automatically set by th...
about_break                       HelpFile                          A statement for immediately exiti...
about_command_search              HelpFile                          How the Windows PowerShell locate...
about_command_syntax              HelpFile                          Command format in the Windows Pow...
about_commonparameters            HelpFile                          Parameters that every cmdlet supp...
about_comparison_operators        HelpFile                          Operators that can be used in the...
about_continue                    HelpFile                          Immediately return to top of a pr...
about_core_commands               HelpFile                          Windows PowerShell core Cmdlets r...
about_display.xml                 HelpFile                          Controlling how objects are displ...
about_environment_variable        HelpFile                          How to access Windows environment...
about_escape_character            HelpFile                          Change how the Windows PowerShell...
about_execution_environment       HelpFile                          Factors that affect how commands run
about_filter                      HelpFile                          Using the Where-Object Cmdlet to ...
about_flow_control                HelpFile                          Using flow control statements in ...
about_for                         HelpFile                          A language command for running a ...
about_foreach                     HelpFile                          A language command for traversing...
about_function                    HelpFile                          Creating and using functions in t...
about_globbing                    HelpFile                          See Wildcard
about_history                     HelpFile                          Retrieving commands entered at th...
about_if                          HelpFile                          A language command for running a ...
about_line_editing                HelpFile                          Editing commands at the Windows P...
about_location                    HelpFile                          Accessing items from the working ...
about_logical_operator            HelpFile                          Operators that can be used in the...
about_method                      HelpFile                          Using methods to perform actions ...
about_namespace                   HelpFile                          Namespaces maintained by the Wind...
about_object                      HelpFile                          Working with objects in the Windo...
```

Figure 1-15

An alternative way to explore the available help files for an install on 32-bit hardware is to open Windows Explorer, navigate to the folder `C:\Windows\System32\WindowsPowerShell\v1.0`, and look for text files whose name begins with `about`. If your system drive is not drive C: modify the path accordingly.

Basic Housekeeping

On the surface, a lot of PowerShell works in the same way as `CMD.exe`. In this section, I describe a couple of basic commands that you will likely use frequently.

To clear the screen, you can type:

```
clear-host
```

or:

```
clear
```

or:

```
cls
```

at the PowerShell command line.

To repeat the last-used PowerShell command, press the F3 key once.

To cycle through recently used PowerShell commands, press the up arrow as necessary to move back to the command that you want to reuse or to adapt. You can also use the `get-history` cmdlet to see the command history. By default, you will be able to see the last 64 commands, but if you or an administrator has modified the value of the `$MaximumHistoryCount` variable, the number of commands available in the history may differ.

At the risk of stating the obvious, PowerShell offers you a number of ways to review information that has scrolled out of sight by using the scroll bars in the PowerShell command window. Click in the scroll bar area or drag the slider in the scroll bar area to move up and down through the information in the PowerShell console window.

Case Insensitivity

In PowerShell, cmdlet names are case-insensitive. In general, cmdlet parameter information is generally also case-insensitive, although there are cases where this is not the case.

All PowerShell cmdlet names, in the *verb-noun* form are case-insensitive. Similarly, all named parameters have parameter names that are case-insensitive. For example, to retrieve information about available commands you can use:

```
get-command
```

or:

```
Get-Command
```

or any other variant of the name using mixed case.

The Windows operating system does not consider case significant in filenames. So, any time that you use a filename as an argument to a PowerShell command, case is not significant by default. For example, to redirect the current date and time to a file named `Text.txt` on drive `C:`, use the following command, which includes redirection:

```
get-date > C:\Test.txt
```

The > character is the redirection operator, which redirects output from the screen (the default) to some specified target — in this case, a file on drive `C:`.

An exception to the general rule of no case-sensitivity is when you use class names from the .NET Framework. PowerShell allows you work directly with classes from the .NET Framework. I discuss this in more detail in Chapter 13.

What You Get in PowerShell

On the surface, PowerShell simply appears to be a new command shell, but you get a highly flexible scripting language with it, too. The following sections describe aspects of the PowerShell package and provide some simple examples of how you can use it.

Interactive Command Shell

As I showed you earlier in the chapter, PowerShell comes complete with a range of commands, called cmdlets, that you can use interactively. By combining these commands in pipelines, you can filter, sort, and group objects. Pipelines are a way of combining commands. They have the general form:

```
command1 | command2
```

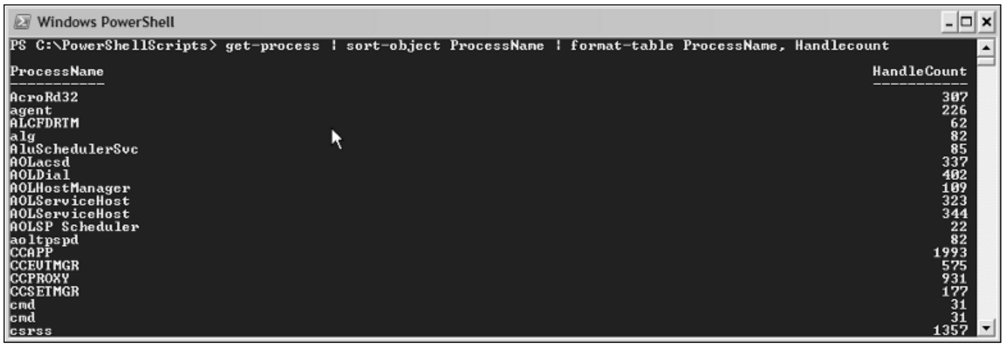
where each step of the pipeline may contain a PowerShell cmdlet, often using multiple parameters. The | character is used to separate the steps of a pipeline. A pipeline can be of arbitrary length.

In the rest of this book, I demonstrate some of the neat tricks you can use to take advantage of pipelines to manage your system.

The following command is a three-step pipeline that retrieves information about running processes, sorts it by process name, and displays selected parts of the results in a table.

```
get-process svchost |
sort-object ProcessName |
format-table ProcessName, Handlecount
```

As you can see in Figure 1-16, you can type the pipeline on a single line. In this book, I will generally present multistep pipelines on multiple lines, since that makes it easier for you to see what each step of the pipeline is doing.

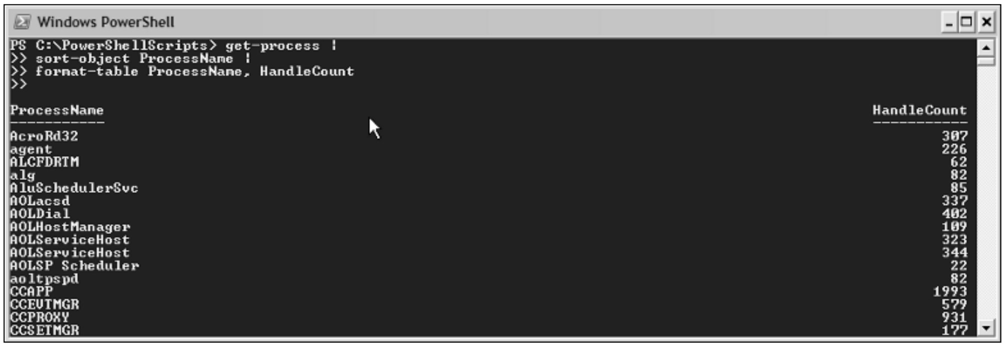


```
PS C:\PowerShellScripts> get-process | sort-object ProcessName | format-table ProcessName, Handlecount

ProcessName                                                                        HandleCount
-----------                                                                        -----------
AcroRd32                                                                                   307
agent                                                                                      226
ALCFDRTM                                                                                    62
alg                                                                                         82
AluSchedulerSvc                                                                             85
AOLacsd                                                                                    337
AOLDial                                                                                    402
AOLHostManager                                                                             109
AOLServiceHost                                                                             323
AOLServiceHost                                                                             344
AOLSP Scheduler                                                                             22
aoltpspd                                                                                    82
CCAPP                                                                                     1993
CCEVTMGR                                                                                   575
CCPROXY                                                                                    931
CCSETMGR                                                                                   177
cmd                                                                                         31
cmd                                                                                         31
csrss                                                                                     1357
```

Figure 1-16

If you prefer, you can type each step of multistep pipelines on separate lines on the command line, which I show you in Figure 1-17. Notice that each step of the pipeline except the last ends in the pipe character (|) and that the command prompt changes to >>. After you type the last step of the pipeline, press Return twice and the command will be executed as if you had typed it all on a single line.



```
PS C:\PowerShellScripts> get-process |
>> sort-object ProcessName |
>> format-table ProcessName, HandleCount
>>

ProcessName                                                                   HandleCount
-----------                                                                   -----------
AcroRd32                                                                              307
agent                                                                                 226
ALCFDRTM                                                                               62
alg                                                                                    82
AluSchedulerSvc                                                                        85
AOLacsd                                                                               337
AOLDial                                                                               402
AOLHostManager                                                                        109
AOLServiceHost                                                                        323
AOLServiceHost                                                                        344
AOLSP Scheduler                                                                        22
aoltpspd                                                                               82
CCAPP                                                                                1993
CCEVTMGR                                                                              579
CCPROXY                                                                               931
CCSETMGR                                                                              177
```

Figure 1-17

Often, you will use the final step of a pipeline to choose how to display information. However, that's not essential, since there is default formatting of output. However, you can use formatting commands as the final step in a pipeline to customize the display and produce a desired output format.

Cmdlets

In a default install of PowerShell version 1, you get more than 100 cmdlets. I look at these individually in more detail in Chapter 4 and later chapters.

If you want to count the number of cmdlets in your version of PowerShell, you can type the following at the command line:

```
$a = get-command;$a.count
```

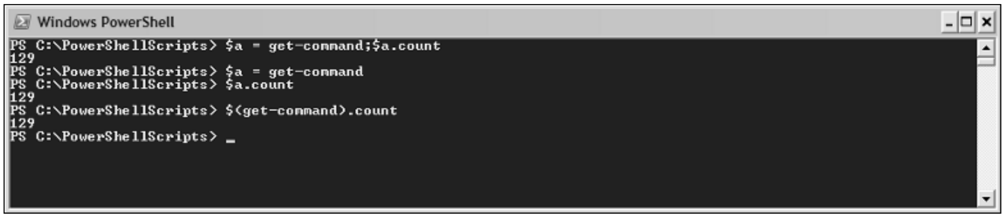
or:

```
$a = get-command
$a.count
```

or:

```
$(get-command).count
```

The semicolon is the separator when you enter multiple PowerShell commands on one line. Alternatively, you can simply enter each PowerShell command on a separate line. As you can see in Figure 1-18, in the version I was using when I wrote this chapter, there were 129 cmdlets available to me. The figure you see may vary significantly, depending on whether additional cmdlets have been installed on your system.



```
PS C:\PowerShellScripts> $a = get-command;$a.count
129
PS C:\PowerShellScripts> $a = get-command
PS C:\PowerShellScripts> $a.count
129
PS C:\PowerShellScripts> $(get-command).count
129
PS C:\PowerShellScripts> _
```

Figure 1-18

The first part of the command is an assignment statement:

```
$a = get-command
```

which assigns all the objects returned by the `get-command` cmdlet to the variable `$a`.

The second part of the command:

```
$a.count
```

uses the `count` property of the variable `$a` to return the number of cmdlets assigned earlier to `$a`. The default output is to the screen so the value of the `count` property is displayed on screen.

Scripting Language

PowerShell provides a new scripting language for the administration of Windows systems. Anything that you type at the command line can be stored as a PowerShell script and reused, as required, at a later date. Often, you will use PowerShell commands in an exploratory way on the command line to define and refine what you want to do. Once you have got things just right, you can store the commands in a PowerShell script and run the script at appropriate times.

In this example, you test a combination of commands on the command line with a view to saving them later as a simple script.

For example, suppose that you want to store in a text file the number of processes running on a machine together with date and time. You could do this by running the following commands, one at a time, on the command line:

1. First, assign to the variable `$a` the result of running the `get-process` cmdlet:

```
$a = get-process
```

2. Then assign to the variable `$b` the value returned from the `get-date` cmdlet:

```
$b = get-date
```

3. Then concatenate a label with the count of processes with the data and time converted to a string and assign the string to the variable `$c`:

```
$c = "Process Count: " + $a.count + " at " + $b.ToString()
```

4. To keep an eye on the current value of `$c`, write it to the host:

```
write-host $c
```

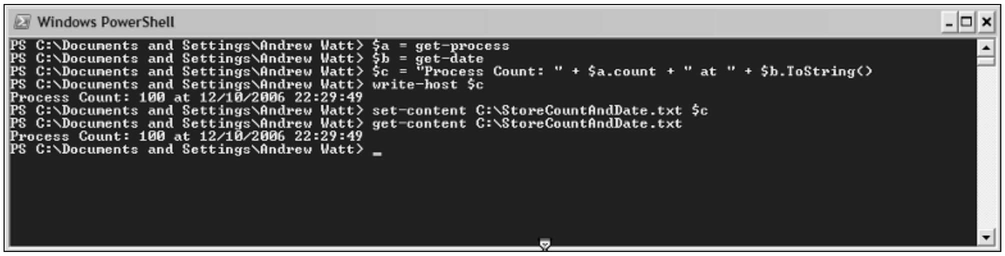
5. Then write the value of $c to a text file:

```
set-content C:\StoreCountAndDate.txt $c
```

6. After doing this, use the following command to show the current information in the text file:

```
get-content C:\StoreCountAndDate.txt
```

The result of this simple exploration is shown in Figure 1-19. The result displayed depends on how many times you have run the commands and at what times.


Windows PowerShell

```
PS C:\Documents and Settings\Andrew Watt> $a = get-process
PS C:\Documents and Settings\Andrew Watt> $b = get-date
PS C:\Documents and Settings\Andrew Watt> $c = "Process Count: " + $a.count + " at " + $b.ToString()
PS C:\Documents and Settings\Andrew Watt> write-host $c
Process Count: 100 at 12/10/2006 22:29:49
PS C:\Documents and Settings\Andrew Watt> set-content C:\StoreCountAndDate.txt $c
PS C:\Documents and Settings\Andrew Watt> get-content C:\StoreCountAndDate.txt
Process Count: 100 at 12/10/2006 22:29:49
PS C:\Documents and Settings\Andrew Watt> _
```

Figure 1-19

The `get-process` command returns all active processes on the machine.

The `get-date` cmdlet returns the current date and time.

You use the `count` property of the variable `$a` to return the number of processes that are active, then use string concatenation and assign that string to `$c`. The `ToString()` method of the `datetime` object converts the date and time to a string.

The `set-content` cmdlet adds information to the specified file. The `get-content` cmdlet retrieves the information contained in the specified text file. The default output is to the screen.

Once you have decided that the individual commands give the desired result — in this case, adding a count of active processes together with a date and time stamp to a selected text file — you can create a script to be run at appropriate times. To run the script, you need to enable script execution as described earlier in this chapter.

The following script, `StoreCountAndDate.ps1`, stores a count of active processes on a machine together with the current `datetime` value.

1. Open Notepad, or your other favorite editor, and type the following code:

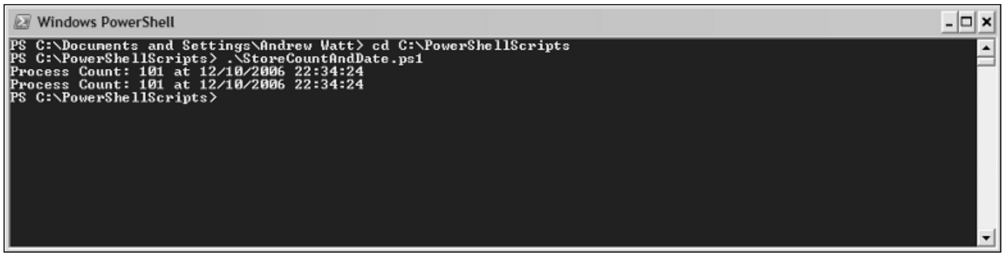
```
$a = get-process
$b = get-date
$c = "Process Count: " + $a.count + " at " + $b.ToString()
write-host $c
set-content C:\StoreCountAndDate.txt $c
get-content C:\StoreCountAndDate.txt
```

2. Save the code in the current folder as `StoreCountAndDate.ps1`. The file extension for PowerShell version 1 scripts is `.ps1`. If you use Notepad, enclose the filename in quotation marks or Notepad will save the code as `StoreCountAndDate.ps1.txt`, which you won't be able to run as a PowerShell script.
3. Run the code by typing:

```
.\StoreCountAndDate
```

at the command line. This works even if the folder has not been added to the `PATH` environment variable.

The result should look similar to Figure 1-20.



```
PS C:\Documents and Settings\Andrew Watt> cd C:\PowerShellScripts
PS C:\PowerShellScripts> .\StoreCountAndDate.ps1
Process Count: 101 at 12/10/2006 22:34:24
Process Count: 101 at 12/10/2006 22:34:24
PS C:\PowerShellScripts>
```

Figure 1-20

To run a PowerShell script in the current directory from the command line, type a period and a back-slash followed by the name of the file (with or without the `ps1` suffix).

If your script is in the current working folder, just typing the script name does *not* work. This is because, with PowerShell, the current working folder (i.e., ".") is not part of the path. To run a script from the current folder, you have to explicity state the folder name. For example: `C:\PowerShellScripts\StoreCountAndDate.ps1` or `.\script.ps1`.

If you have difficulty running the script, it may be that running unsigned PowerShell scripts is not allowed on the computer you are using. If you follow the suggestion earlier in this chapter to set the execution policy to `RemoteSigned`, the script should run. Alternatively, you will need to sign the script in a way acceptable to your organization. I discuss signing scripts in Chapter 15

Summary

Windows PowerShell is a new command shell and scripting language for the Windows platform. This chapter showed you how to install the .NET Framework 2.0 and how to install Windows PowerShell.

In this chapter, you also learned how to carry out the following tasks:

- ❑ Start PowerShell
- ❑ Exit PowerShell
- ❑ Find out what PowerShell commands are available on your system
- ❑ Get help on individual PowerShell commands
- ❑ Develop and run a simple PowerShell script

You can, of course, create much more complex scripts in PowerShell than the example I showed in this chapter. Before going on, in Chapter 4, to begin to look in more detail at how some individual cmdlets can be used, I will step aside in Chapter 2 to look at the broader of issues of what is lacking in existing approaches and in Chapter 3 go on to look at the Windows PowerShell approach to improving on what was previously available.