Get started with this powerful Windows administration tool

Automate Windows administration tasks with ease by learning the fundamentals of Windows PowerShell 3.0. Led by a Windows PowerShell expert, you’ll learn must-know concepts and techniques through easy-to-follow explanations, examples, and exercises. Once you complete this practical introduction, you can go deeper into the Windows PowerShell command line interface and scripting language with *Windows PowerShell 3.0 Step by Step.*

Discover how to:

- Create effective Windows PowerShell commands with one line of code
- Apply Windows PowerShell commands across several Windows platforms
- Identify missing hotfixes and service packs with a single command
- Sort, group, and filter data using the Windows PowerShell pipeline
- Create users, groups, and organizational units in Active Directory
- Add computers to a domain or workgroup with a single line of code
- Run Windows PowerShell commands on multiple remote computers
- Unleash the power of scripting with Windows Management Instrumentation (WMI)

About the Author

**Ed Wilson**, a senior consultant at Microsoft Corporation, is a scripting expert who delivers workshops to Microsoft customers and employees worldwide. His books on Windows scripting include *Windows PowerShell 2.0 Best Practices* and *Microsoft VBScript Step by Step*. Ed also writes the popular TechNet blog, “Hey, Scripting Guy!”
Windows PowerShell 3.0
First Steps

Ed Wilson
To Teresa, my soul mate.

—Ed Wilson
## Contents at a glance

*Foreword*  
*Introduction*  

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview of Windows PowerShell 3.0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Using Windows PowerShell cmdlets</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>Filtering, grouping, and sorting</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>Formatting output</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>Storing output</td>
<td>69</td>
</tr>
<tr>
<td>6</td>
<td>Leveraging Windows PowerShell providers</td>
<td>79</td>
</tr>
<tr>
<td>7</td>
<td>Using Windows PowerShell remoting</td>
<td>99</td>
</tr>
<tr>
<td>8</td>
<td>Using WMI</td>
<td>113</td>
</tr>
<tr>
<td>9</td>
<td>Using CIM</td>
<td>127</td>
</tr>
<tr>
<td>10</td>
<td>Using the Windows PowerShell ISE</td>
<td>141</td>
</tr>
<tr>
<td>11</td>
<td>Using Windows PowerShell scripts</td>
<td>153</td>
</tr>
<tr>
<td>12</td>
<td>Working with functions</td>
<td>183</td>
</tr>
<tr>
<td>13</td>
<td>Debugging scripts</td>
<td>203</td>
</tr>
<tr>
<td>14</td>
<td>Handling errors</td>
<td>217</td>
</tr>
<tr>
<td>A</td>
<td>Windows PowerShell FAQ</td>
<td>229</td>
</tr>
<tr>
<td>B</td>
<td>Windows PowerShell 3.0 coding conventions</td>
<td>239</td>
</tr>
</tbody>
</table>

*Index*  

247
# Contents

*Foreword* xv

*Introduction* xvii

## Chapter 1  Overview of Windows PowerShell 3.0

1

Understanding Windows PowerShell 1

Working with Windows PowerShell 2

Security issues with Windows PowerShell 4

Using Windows PowerShell cmdlets 6

The most common verb: *Get* 6

Supplying options for cmdlets 12

Using single parameters 13

Introduction to parameter sets 16

Using command-line utilities 18

Working with Help options 19

Summary 20

## Chapter 2  Using Windows PowerShell cmdlets

21

Understanding the basics of cmdlets 22

Common Windows PowerShell parameters 22

Starting the Windows PowerShell transcript 24

Stopping and reviewing the Windows PowerShell transcript 25

Searching the Help topics 26

Using the *Get-Help* cmdlet 26

Using the About conceptual Help topics 29

---

**What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

Using the *Get-Command* to find cmdlets .................................................. 30
Using the *Get-Member* cmdlet ................................................................. 33
  Exploring property members .............................................................. 34
Using the *Show-Command* cmdlet ......................................................... 34
Setting the Script Execution Policy ......................................................... 36
Creating a basic Windows PowerShell profile ........................................ 37
  Determining if a Windows PowerShell profile exists .......................... 38
  Creating a new Windows PowerShell profile ...................................... 38
Summary ................................................................................................. 39

**Chapter 3**  **Filtering, grouping, and sorting**  41
Introduction to the pipeline ................................................................. 41
Sorting output from a cmdlet .............................................................. 42
Grouping output after sorting ............................................................. 44
  Grouping information without element data ................................. 45
Filtering output from one cmdlet ......................................................... 46
  Filtering by date ................................................................................. 47
  Filtering to the left .......................................................................... 49
Filtering output from one cmdlet before sorting .............................. 50
Summary ................................................................................................. 51

**Chapter 4**  **Formatting output**  53
Creating a table ...................................................................................... 53
  Choosing specific properties in a specific order .............................. 54
  Controlling the way the table displays ............................................ 55
Creating a list ......................................................................................... 58
  Choosing properties by name .......................................................... 59
  Choosing properties by wildcard ..................................................... 59
Creating a wide display ....................................................................... 61
  Using the `-AutoSize` parameter to configure the output ............. 61
  Customizing the *Format-Wide* output ........................................... 62
## Contents

Creating an output grid ........................................... 63
  Sorting output by using the column buttons 64
  Filtering output by using the filter box 66
Summary ................................................................. 67

### Chapter 5  Storing output 69

Storing data in text files ........................................ 69
  Redirect and append 70
  Redirect and overwrite 71
  Controlling the text file 72
Storing data in .csv files ........................................ 73
  No type information 73
  Using type information 75
Storing data in XML ................................................. 76
  The problem with complex objects 76
  Using XML to store complex objects 76
Summary ................................................................. 78

### Chapter 6  Leveraging Windows PowerShell providers 79

Understanding Windows PowerShell providers ................. 80
  Understanding the Alias provider 80
  Understanding the Certificate provider 82
  Understanding the Environment provider 85
  Understanding the File System provider 86
  Understanding the Function provider 88
  Understanding the Registry provider 89
  Understanding the Variable provider 96
Summary ................................................................. 97

### Chapter 7  Using Windows PowerShell remoting 99

Using Windows PowerShell remoting ............................ 99
  Classic remoting 99
Chapter 11 Using Windows PowerShell scripts

Why write Windows PowerShell scripts?............................ 153

Scripting fundamentals ............................................. 155
  Running Windows PowerShell scripts 155
  Enabling Windows PowerShell scripting support 156
  Transitioning from command line to script 157
  Running Windows PowerShell scripts 159
  Understanding variables and constants 160

Using the While statement ........................................ 162
  Constructing the While statement 162
  A practical example of using the While statement 164
  Using special features of Windows PowerShell 164

Using the Do...While statement ............................... 165
  Using the range operator 166
  Operating over an array 166
  Casting to ASCII values 167

Using the Do...Until statement ................................... 168
  Using the Windows PowerShell Do...Loop statement 168

Using the For statement ............................................ 170
  Creating a For...Loop 170
  Using the ForEach statement 172
  Exiting the ForEach statement early 174

Using the If statement .............................................. 175
  Using assignment and comparison operators 177
  Evaluating multiple conditions 178
Using the *Switch* statement ........................................... 179
  Using the basic *Switch* statement 180
  Controlling matching behavior 182
Summary ............................................................................. 182

**Chapter 12 Working with functions** 183

Understanding functions .............................................. 183
  Using a type constraint 190
Using multiple input parameters ................................. 192
Using functions to encapsulate business logic ................. 194
Using functions to provide ease of modification .............. 196
Summary ............................................................................. 201

**Chapter 13 Debugging scripts** 203

Understanding debugging in Windows PowerShell .......... 203
Debugging the script ...................................................... 203
  Setting breakpoints 204
  Setting a breakpoint on a line number 204
  Setting a breakpoint on a variable 206
  Setting a breakpoint on a command 209
  Responding to breakpoints 211
  Listing breakpoints 213
  Enabling and disabling breakpoints 215
  Deleting breakpoints 215
Summary ............................................................................. 216

**Chapter 14 Handling errors** 217

Handling missing parameters ........................................ 217
  Creating a default value for the parameter 218
  Making the parameter mandatory 219
Foreword

There are many reasons to get started with automation. For me it was a little turtle from a program called LOGO. Of course, at the time I had no idea I was learning programming. I was just a kid in elementary school having fun, drawing little pictures. Years later, I became an IT administrator and developed an aversion to tedious tasks, such as manually copying a file to 100 remote servers. I started automating because I just couldn’t stand the thought of repeating monotonous tasks over and over again. It took a while before I connected the dots and realized that the little turtle had paved the way for a career focused on using and teaching automation.

Windows PowerShell has really hit a sweet spot with automation in the Windows universe, balancing powerful and far-reaching capabilities while remaining simple enough that someone without deep technical expertise can start taking advantage of it quickly. Though Windows PowerShell can be a simple automation environment, it has nuances that can make it a bit tricky to really master, akin to driving a car with a manual transmission. It might be tricky to get started, but once the car is moving in first gear, the rest comes pretty easily. Ed Wilson has done a wonderful job in this book getting you started in Windows PowerShell, providing simple, prescriptive guidance to get you into first gear quickly.

As a Senior Premier Field Engineer and a Windows PowerShell Technology Lead for Microsoft Services, I spend most of my days in front of Microsoft’s customers trying to teach them Windows PowerShell and hopefully getting them to love Windows PowerShell as much as I do. In every class I teach, I can’t stress enough the return on investment (ROI) you get from learning Windows PowerShell. It never ceases to amaze me how once you grasp the core concepts of Windows PowerShell, you can apply them over and over again to get so much business value and personal satisfaction.

One point I try to make during every class I teach is that the words “Windows PowerShell” and “scripting” can most definitely be mutually exclusive. Technically speaking, Windows PowerShell one-liners are still “scripts,” but to me they strike a nice balance between the creation of solutions and the need for developer-oriented skills. One-liners are usually very task-oriented and logically simple, yet they can accomplish a staggering amount of automation. Those who are just getting started with Windows PowerShell will find that they can become great at Windows PowerShell without writing scripts. Throughout much of this book, Ed has focused on the concepts and simplicity of Windows PowerShell. He doesn’t talk directly about scripting until late in the book. Ultimately, scripting and tool-making become parts of the advanced user’s skill set, but you can go a long way before that needs to happen.

No matter how diverse the skill set of my students, there is something for everyone in my classroom. Windows PowerShell has been created in such a way that it can be fun and effective for everyone from the IT novice to the expert developer. For example, the fact that it is
fully object-based and sits on top of the .NET Framework is a detail that pure beginners might have no knowledge of. They can go about their Windows PowerShell days simply running commands, never really digging into the object model, but still implement valuable automation. The day they learn about objects, they can start to unlock so much more. The fact that Windows PowerShell can appeal to such diverse skills levels simultaneously is amazing to me.

When I really think about the value of Windows PowerShell and why someone new to it should dive right in, I think about the fundamental comparison of “creation” vs. “operation.” By over-simplifying the roles in IT, you can see a dividing line between developers and administrators. Developers are creating solutions, and administrators are managing the design, deployment, and operation of the systems used in the process. Windows PowerShell can bridge the dividing gap to link it all together. It also allows administrators to create automation solutions without needing a true developer. There are enough elements in the Windows PowerShell language that hide and simplify the true complexity that lurks under the surface, allowing IT pros to be more effective and valuable in the workplace. Learning Windows PowerShell is an incredibly powerful tool that will truly make you more valuable to your business and often make your life easier in the process.

Ed “The Scripting Guy” Wilson is what some people call a “PowerShellebrity.” He’s a superstar in the Windows PowerShell world, has extensive scripting experience, and is one of the most energetic and passionate people I have ever met. I am grateful that Ed writes these books because it allows so many people access to his extensive experience and knowledge. This book is such a concise and easy way to get started with Windows PowerShell, I can’t imagine putting it down if I were a beginner. Whether you have already started your Windows PowerShell journey or are just getting started, this book will help define your next steps with Windows PowerShell.

—Gary Siepsert
Senior Premier Field Engineer (PFE)
Microsoft Corporation
Introduction

Gary said nearly everything I wanted to include in the Introduction. I designed this book for the complete beginner, and you should therefore read the book from beginning to end. If you want a more reference oriented book, you should check out my PowerShell Best Practices books, or even PowerShell 3.0 Step by Step. Actually, the Step by Step book is not really a reference, but a hands-on learning guide. It is, ideally, the book you graduate to once you have completed this one. For your daily dose of PowerShell, you should check out my Hey Scripting Guy blog at www.ScriptingGuys.com/blog. I post new content there twice a day.

System Requirements

Hardware Requirements
Your computer should meet the following minimum hardware requirements:
- 2.0 GB of RAM (more is recommended)
- 80 GB of available hard disk space
- Internet connectivity

Software Requirements
To complete the exercises in this book, you should have Windows PowerShell 3.0 installed:
- You can obtain Windows PowerShell 3.0 from the Microsoft Download Center by downloading the Windows Management Framework and installing it on either Windows 7 Service Pack 1, Windows Server 2008 R2 SP1, or Windows Server 2008 Service Pack 2.
- The section on Active Directory requires access to Active Directory Domain Services. For those examples, ensure you have access to Windows Server 2012.
Acknowledgments

Many people contributed the success of this book. The first person is Teresa Wilson, aka "The Scripting Wife." She is always my first reader, and nothing leaves the house without her approval. Second, I must mention my tech reviewer, Brian Wilhite, who did a great job of catching bugs, errors, and things that are misleading. I also want to thank the Charlotte PowerShell User Group whose questions, comments, and the like contributed in a significant way to the book. I kept them in mind as I wrote. I also want to thank Michael Bolinger and Melanie Yarbrough from O'Reilly for doing a great job seeing this project to completion.

Support & Feedback

The following sections provide information on errata, book support, feedback, and contact information.

Errata
We have made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://aka.ms/WinPS3FS/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, please email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You
At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch
Let us keep the conversation going! We are on Twitter: http://twitter.com/MicrosoftPress
Overview of Windows PowerShell 3.0

- Understanding Windows PowerShell
- Working with Windows PowerShell
- Using Windows PowerShell cmdlets
- Supplying options for cmdlets
- Working with Help options

When you first start Windows PowerShell, whether it is the Windows PowerShell console or the Windows PowerShell Integrated Scripting Environment (ISE), the blank screen simply waits for your command. The problem is there are no hints as to what that command might be. There are no wizards or other Windows types of features to guide you in using the shell.

The name is Windows PowerShell for two reasons: It is a shell, and it is powerful. It is a mistake to think that Windows PowerShell is simply a scripting language because it is much more than that. In the same way, it is a mistake to think that Windows PowerShell is limited to running only a few cmdlets. Through scripting, it gains access to the entire realm of management technology available in the Windows world.

This chapter introduces you to Windows PowerShell and illustrates the incredible power available to you from this flexible and useful management tool.

Understanding Windows PowerShell

Windows PowerShell comes in two flavors. The first is an interactive console (similar to a KORN or BASH console in the UNIX world) built into the Windows command prompt. The Windows PowerShell console makes it simple to type short commands and to receive sorted, filtered, and formatted results. These results easily display to the console but also can redirect to .xml, .csv, or text files. The Windows PowerShell console offers several advantages such as speed, low memory overhead, and a comprehensive transcription service that records all commands and command output.
The other flavor of Windows PowerShell is the Windows PowerShell ISE. The Windows PowerShell ISE is an Integrated Scripting Environment, but this does not mean you must use it to write scripts. In fact, many Windows PowerShell users like to write their code in the Windows PowerShell ISE to take advantage of syntax coloring, drop-down lists, and automatic parameter revelation features.

In addition, the Windows PowerShell ISE has a feature called Show Command Add-On that allows you to use a mouse to create Windows PowerShell commands from a graphical environment. Once you create the command, the command either runs directly or is added to the Script pane. The choice is up to you. For more information about using the Windows PowerShell ISE, see Chapter 10, “Using the Windows PowerShell ISE.”

**NOTE** When I work with single commands, for simplicity I show the command and results from within the Windows PowerShell console. But keep in mind that all commands also run from within the Windows PowerShell ISE. Whether the command runs in the Windows PowerShell console, in the Windows PowerShell ISE, as a scheduled task, or as a filter for Group Policy, Windows PowerShell is Windows PowerShell is Windows PowerShell. In its most basic form, a Windows PowerShell script is simply a collection of Windows PowerShell commands.

**Working with Windows PowerShell**

Windows PowerShell 3.0 is included on Windows 8 and Windows Server 2012. On Windows 8, you need only type the first few letters of the word PowerShell in the Start window before Windows PowerShell appears as an option. Figure 1-1 illustrates this point. I typed only pow in the Search box before the Start window offered Windows PowerShell as an option.
Because navigating to the Start window and typing `pow` each time I want to launch Windows PowerShell is a bit cumbersome, I prefer to pin shortcuts to the Windows PowerShell console (and the Windows PowerShell ISE) to both the Start window and the Windows taskbar. This technique of pinning shortcuts to the applications, as shown in Figure 1-2, provides single-click access to Windows PowerShell from wherever I might be working.

**FIGURE 1-1** Typing in the Start window opens the Search window highlighting the Windows PowerShell console.

**FIGURE 1-2** By right-clicking the Windows PowerShell icon in the Search results box, the Pin to Start and the Pin to taskbar options appear.
On Windows Server 2012, it is unnecessary to find the icon by using the Search box on the Start window because an icon for the Windows PowerShell console exists by default on the taskbar of the desktop.

**NOTE** The Windows PowerShell ISE (the script editor) does not exist by default on Windows Server 2012. You need to add the Windows PowerShell ISE as a feature. I show how to use the Windows PowerShell ISE in Chapter 10, “Using the Windows PowerShell ISE.”

**Security issues with Windows PowerShell**

There are two ways to launch Windows PowerShell: as an administrator or as a normal, or non-elevated, user. As a best practice, start Windows PowerShell with minimum rights. On Windows 7 and Windows 8, this means simply clicking on the Windows PowerShell icon. It opens as a non-elevated user, even if you are logged on with administrator rights. On Windows Server 2012, Windows PowerShell automatically launches with the rights of the current user. Therefore, if you are logged on as a domain administrator, the Windows PowerShell console launches with domain administrator rights.

**Running as a non-elevated user**

Because Windows PowerShell adheres to Windows security constraints, a user of Windows PowerShell cannot do anything the user account does not have permission to do. Therefore, if you are a non-elevated user, you do not have rights to perform tasks such as installing printer drivers, reading from the Security Log, or changing the system time.

If you are an administrator on a local Windows 7 or Windows 8 computer and you do not launch Windows PowerShell with administrator rights, you will get errors when you attempt to take certain actions, such as viewing the configuration of your disk drives. The following example shows the command and associated error:

```
PS C:\> get-disk
get-disk : Access to a CIM resource was not available to the client.
At line:1 char:1
+ get-disk
+ ~~~~~~~~
   + CategoryInfo : PermissionDenied: (MSFT_Disk:ROOT/Microsoft/Windows/Storage/MSFT_Disk) [Get-Disk], CimException
   + FullyQualifiedErrorId : MI RESULT 2,Get-Disk
```
If you attempt to run cmdlets that require elevated rights, you will encounter inconsistencies with errors. For example, in a non-elevated Windows PowerShell console, the error from the `Get-Disk` cmdlet is **Access To A CIM Resource Was Not Available To The Client.** The error from the `Stop-Service` cmdlet is **Cannot Open XXX Service On Computer.** The `Get-VM` cmdlet simply returns no information and no error. Therefore, check for console rights as a first step in troubleshooting.

**Launching Windows PowerShell with administrator rights**

To perform tasks that require administrator rights, you must start the Windows PowerShell console with administrator rights. To do this, right-click the Windows PowerShell icon (the one pinned to the taskbar, the one on the Start window, or the one found by using the Search box in the Start window) and select the Run As Administrator option from the Action menu. The great advantage of this technique is that you can launch either the Windows PowerShell console (the first item on the menu) as an administrator, or from the same screen you can launch the Windows PowerShell ISE as an administrator. Figure 1-3 shows these options.

Once you launch the Windows PowerShell console with administrator rights, the User Account Control (UAC) dialog box appears, requesting permission to allow Windows PowerShell to make changes to the computer. In reality, Windows PowerShell is not making changes to the computer, at least not yet. But using Windows PowerShell, you can certainly make changes to the computer if you have the rights. This is what the dialog box is prompting you for.

**NOTE** It is possible to avoid this prompt by turning off UAC. However, UAC is a significant security feature, so I do not recommend disabling it. The UAC has been fine-tuned on Windows 7 and Windows 8. The number of UAC prompts has been greatly reduced from the number that used to exist with the introduction of UAC on Windows Vista.
Now that you are running Windows PowerShell with administrator rights, you can do anything your account has permission to do. For example, if you run the Get-Disk cmdlets, you will see information similar to the following:

```
PS C:\> get-disk

Number Friendly Name          Operational status Total Size  Partition Style
------- -------------                            ------------  ----------  -----------
0 INTEL SSDSA2BW160G3L   Online               149.05 GB MBR
```

### Using Windows PowerShell cmdlets

Windows PowerShell cmdlets all work in a similar fashion. This simplifies their use. All Windows PowerShell cmdlets have a two-part name. The first part is a verb, although the verb is not always strictly grammatical. The verb indicates the action for the command to take. Examples of verbs include Get, Set, Add, Remove, and Format. The noun is the thing to which the action will apply. Examples of nouns include Process, Service, Disk, and NetAdapter. A dash combines the verb with the noun to complete the Windows PowerShell command. Windows PowerShell commands are named cmdlets (pronounced command let) because they behave like small commands or programs that are used standalone or pieced together through a mechanism called the **pipeline**. For more information about the pipeline, see Chapter 2, “Using Windows PowerShell Cmdlets.”

### The most common verb: Get

Out of nearly 2,000 cmdlets (and functions) on Windows 8, over 25 percent of them use the verb Get. The verb Get retrieves information. The noun portion of the cmdlet specifies the information retrieved. To obtain information about the processes on your system, open the Windows PowerShell console by either clicking the Windows PowerShell icon on the taskbar or typing **PowerShell** on the Start window of Windows 8 to bring up the search results for Windows PowerShell, as discussed in a preceding section, “Launching Windows PowerShell with administrator rights.”

Once the Windows PowerShell console appears, run the Get-Process cmdlet. To do this, use the Windows PowerShell Tab Completion feature to complete the cmdlet name. Once the cmdlet name appears, press the Enter key to cause the command to execute.
NOTE The Windows PowerShell Tab Completion feature is a great time saver. It not only saves time by reducing the need for typing, but it also helps to ensure accuracy because tab completion accurately resolves cmdlet names. It is like a spelling checker for cmdlet names. For example, attempting to type a lengthy cmdlet name such as `Get-NetAdapter-EncapsulatedPacketTaskOffload` accurately could be an exercise in frustration. But if you use tab completion, you have to type only `Get-Net` and press the Tab key about six times before the correctly spelled cmdlet name appears in the Windows PowerShell console. Learning how to quickly and efficiently use tab completion is one of the keys to success for using Windows PowerShell.

Finding process information

To use the Windows PowerShell Tab Completion feature to enter the `Get-Process` cmdlet name at the Windows PowerShell console command prompt, type the following on the first line of the Windows PowerShell console, then press the Tab key followed by Enter:

```
Get-Pro
```

This order of commands—command followed by Tab and Enter—is called *tab expansion*. Figure 1-4 shows the `Get-Process` command and associated output.

![Figure 1-4: The Windows PowerShell Get-Process cmdlet returns detailed Windows process information.](image-url)
To find information about Windows services, use the verb *Get* and the noun *Service*. In the Windows PowerShell console, type the following, then press the Tab key followed by Enter:

```
Get-Servi
```

**NOTE** It is a Windows PowerShell convention to use singular nouns. While not universally applied (my computer has about 50 plural nouns), it is a good place to start. So if you are not sure if a noun (or parameter) is singular or plural, choose the singular. Most of the time you will be correct.

### Identifying installed Windows hotfixes

To find a listing of Windows hotfixes applied to the current Windows installation, use the `Get-Hotfix` cmdlet. The verb is *Get* and the noun is *Hotfix*. In the Windows PowerShell console, type the following, then press the Tab key followed by Enter:

```
Get-Hotf
```

Figure 1-5 shows the `Get-Hotfix` command and associated output.

![Figure 1-5](image)

**FIGURE 1-5** Use the `Get-Hotfix` cmdlet to obtain a detailed listing of all applied Windows hotfixes.
Getting detailed service information

To find information about services on the system, use the Get-Service cmdlet. Once again, it is not necessary to type the entire command. The following command uses tab expansion to complete the Get-Service command and execute it:

Get-Service

NOTE The efficiency of tab expansion depends upon the number of cmdlets, functions, or modules installed on the computer. As more commands become available, the efficiency of tab expansion reduces correspondingly.

The following (truncated) output appears following the Get-Service cmdlet:

PS C:\> Get-Service

<table>
<thead>
<tr>
<th>Status</th>
<th>Name</th>
<th>DisplayName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>AdobeActiveFile...</td>
<td>Adobe Active File Monitor V6</td>
</tr>
<tr>
<td>Stopped</td>
<td>AeLookupSvc</td>
<td>Application Experience</td>
</tr>
<tr>
<td>Stopped</td>
<td>ALG</td>
<td>Application Layer Gateway Service</td>
</tr>
<tr>
<td>Stopped</td>
<td>AllUserInstallA...</td>
<td>Windows All-User Install Agent</td>
</tr>
</tbody>
</table>

Identifying installed network adapters

To find information about network adapters on your Windows 8 or Windows Server 2012 machine, use the Get-NetAdapter cmdlet. Using tab expansion, type the following then press Tab, followed by Enter:

Get-NetAd

The following example shows the command and associated output:

PS C:\> Get-NetAdapter

<table>
<thead>
<tr>
<th>Name</th>
<th>InterfaceDescription</th>
<th>ifIndex</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Bridge</td>
<td>Microsoft Network Adapter Multiplexo...</td>
<td>29</td>
<td>Up</td>
</tr>
<tr>
<td>Ethernet</td>
<td>Intel(R) 82579LM Gigabit Network Controller</td>
<td>13</td>
<td>Not Pre...</td>
</tr>
<tr>
<td>vEthernet (WirelessSwi...)</td>
<td>Hyper-V Virtual Ethernet Adapter #4</td>
<td>31</td>
<td>Up</td>
</tr>
<tr>
<td>vEthernet (External Swi...)</td>
<td>Hyper-V Virtual Ethernet Adapter #3</td>
<td>23</td>
<td>Not Pre...</td>
</tr>
<tr>
<td>vEthernet (InternalSwi...)</td>
<td>Hyper-V Virtual Ethernet Adapter #2</td>
<td>19</td>
<td>Up</td>
</tr>
<tr>
<td>Bluetooth Network Conn...</td>
<td>Bluetooth Device (Personal Area Network Monitor...)</td>
<td>15</td>
<td>Disconnect</td>
</tr>
<tr>
<td>Wi-Fi</td>
<td>Intel(R) Centrino(R) Ultimate-N 6300...</td>
<td>12</td>
<td>Up</td>
</tr>
</tbody>
</table>
Retrieving detected network connection profiles

If you want to see the network connection profile that Windows 8 or Windows Server 2012 detected for each interface, use the `Get-NetConnectionProfile` cmdlet. To run this command, use the following command with tab expansion:

```
Get-NetC
```

The following example shows the command and associated output:

```
PS C:\> Get-NetConnectionProfile

Name             : Unidentified network
InterfaceAlias   : vEthernet (InternalSwitch)
InterfaceIndex   : 19
NetworkCategory  : Public
IPv4Connectivity : NoTraffic
IPv6Connectivity : NoTraffic

Name             : Network 10
InterfaceAlias   : vEthernet (WirelessSwitch)
InterfaceIndex   : 31
NetworkCategory  : Public
IPv4Connectivity : Internet
IPv6Connectivity : NoTraffic
```

**NOTE**  Windows PowerShell is not case sensitive. There are a few instances where case sensitivity is an issue (for example, when using regular expressions) but cmdlet names, parameters, and values are not case sensitive. Windows PowerShell convention uses a combination of uppercase and lowercase letters, generally at syllable breaks in long noun names such as `NetConnectionProfile`. However, this is not a requirement for Windows PowerShell to interpret the command accurately. This combination of uppercase and lowercase letters is for readability. If you use tab expansion, Windows PowerShell automatically converts the commands to this format.

Getting the current culture settings

A typical Windows computer has two categories of culture settings. The first category contains the culture settings for the current culture settings, which includes information about the keyboard layout and the display format of items such as numbers, currency, and dates. To find the value of these cultural settings, use the `Get-Culture` cmdlet. To call the `Get-Culture` cmdlet using tab expansion to complete the command, type the following at the command prompt of the Windows PowerShell console, then press the Tab key followed by Enter:

```
Get-Cu
```

When the command runs basic information such as the Language Code ID number (LCID), the name of the culture settings, in addition to the display name of the culture settings,
return to the Windows PowerShell console. The following example shows the command and associated output:

```powershell
PS C:\> Get-Culture

LCID   Name       DisplayName
-----   ----       -----------
1033    en-US      English (United States)
```

The second category is the current user interface (UI) settings for Windows. The UI culture settings determine which text strings appear in user interface elements such as menus and error messages. To determine the current UI culture settings that are active, use the `Get-UICulture` cmdlet. Using tab expansion to call the `Get-UICulture` cmdlet, type the following, then press the Tab key followed by Enter:

```powershell
Get-UICulture
```

The following example shows the command and associated output:

```powershell
PS C:\> Get-UICulture

LCID   Name       DisplayName
-----   ----       -----------
1033    en-US      English (United States)
```

**NOTE** On my computer, both the current culture and the current UI culture are the same. This is not always the case, and at times I have seen a computer have issues when the user interface is set for a localized language while the computer itself is set for U.S. English. This is especially problematic when using virtual machines created in other countries. In this case, even a simple task such as typing in a password becomes very frustrating. To fix these types of situations, you can use the `Set-Culture` cmdlet.

### Finding the current date and time

To find the current date or time on the local computer, use the `Get-Date` cmdlet. Tab expansion does not help much for this cmdlet because there are 15 cmdlets (on my computer) that have a cmdlet name that begins with the letters `Get-Da`. This includes all the Direct Access cmdlets as well as the Remote Access cmdlets. Therefore, using tab expansion to get the date requires me to type the following before pressing the Tab key followed by the Enter key:

```powershell
Get-Dat
```

The preceding command syntax is the same number of keys to press as the following combined with the Enter key:

```powershell
Get-Date
```
The following example shows the command and associated output:

```
PS C:\> Get-Date

Tuesday, November 20, 2012 9:54:21 AM
```

**Generating a random number**

Windows PowerShell 2.0 introduced the `Get-Random` cmdlet, and when I saw it I was not too impressed at first because I already knew how to generate a random number. As shown in the following example, I can use the .NET Framework `System.Random` class to create a new instance of the `System.Random` object and call the `next` method:

```
PS C:\> (New-Object system.random).next()
225513766
```

 Needless to say, I did not create many random numbers. Who wants to do all that typing? But once I had the `Get-Random` cmdlet, I actually began using random numbers for all sorts of actions. For example, I have used the `Get-Random` cmdlet to do the following:

- Pick prize winners for the Scripting Games.
- Pick prize winners for Windows PowerShell user group meetings.
- Connect to remote servers in a random way for load-balancing purposes.
- Create random folder names.
- Create temporary users in Active Directory with random names.
- Wait a random amount of time prior to starting or stopping processes and services (great for performance testing).

The `Get-Random` cmdlet has turned out to be one of the more useful cmdlets. To generate a random number in the Windows PowerShell console using tab expansion, type the following on the first line in the console, then press the Tab key followed by the Enter key:

```
Get-R
```

The following example shows the command and associated output:

```
PS C:\> Get-Random
248797593
```

**Supplying options for cmdlets**

The easiest Windows PowerShell cmdlets to use require no options. Unfortunately, that is only a fraction of the total number of cmdlets (and functions) available in Windows PowerShell 3.0 as it exists on either Windows 8 or Windows Server 2012. Fortunately, the same tab expansion technique used to create the cmdlet names on the Windows PowerShell console works with parameters as well.
Using single parameters

When working with Windows PowerShell cmdlets, often the cmdlet requires only a single parameter to filter out the results. If a parameter is the default parameter, you do not have to specify the parameter name; you can use the parameter positionally. This means that the first value appearing after the cmdlet name is assumed to be a value for the default (or position 1) parameter. On the other hand, if a parameter is a named parameter, the parameter name (or parameter alias or partial parameter name) is always required when using the parameter.

Finding specific types of hotfixes

To find all the Windows Update hotfixes, use the Get-HotFix cmdlet with the -Description parameter and supply a value of update to the -Description parameter. This is actually easier than it sounds. Once you type Get-Hot and press the Tab key, you have the Get-Hotfix portion of the command. Then a space and -D + Tab completes the Get-HotFix -Description portion of the command. Now you need to type Update and press Enter. With a little practice, using tab expansion becomes second nature.

Figure 1-6 shows the Get-Hotfix command and associated output.

![PowerShell 3](image)

PS C:\> Get-HotFix -Description update

<table>
<thead>
<tr>
<th>Source</th>
<th>Description</th>
<th>HotFixID</th>
<th>InstalledBy</th>
<th>InstalledOn</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2712101</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>10/9/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2693643</td>
<td>IAMMED\ed</td>
<td>10/22/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB27511352</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>9/23/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2758672</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>10/9/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2758944</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>10/9/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2761034</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>10/9/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2764870</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>10/9/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2758703</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>10/9/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2759034</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>11/17/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2770041</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>11/7/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2770407</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>11/17/2012 12:00...</td>
</tr>
<tr>
<td>EDTL</td>
<td>Update</td>
<td>KB2700002</td>
<td>NT AUTHORITY\SYSTEM</td>
<td>11/17/2012 12:00...</td>
</tr>
</tbody>
</table>

**FIGURE 1-6** Add the -Description parameter to the Get-HotFix cmdlet to see specific hotfixes such as updates in a filtered list.

If you attempt to find only update types of hotfixes by supplying the value update in the first position, an error appears. The following example shows the offending command and associated error:

PS C:\> Get-HotFix update
Get-HotFix : Cannot find the requested hotfix on the 'localhost' computer. Verify the input and run the command again.
At line:1 char:1
+ Get-HotFix update
+ ~~~~~~~~~~~~~~~~~
  + CategoryInfo : ObjectNotFound: (:) [Get-HotFix], ArgumentException
The error, while not really clear, seems to indicate that the *Get-HotFix* cmdlet attempts to find a hotfix named update. This is, in fact, the attempted behavior. The Help file information for the *Get-HotFix* cmdlet reveals that `-ID` is position 1, as shown in the following example:

```
-Id <String[]>
  Gets only hotfixes with the specified hotfix IDs. The default is all hotfixes on the computer.

  Required? false
  Position? 1
  Default value All hotfixes
  Accept pipeline input? false
  Accept wildcard characters? False
```

You might ask, “What about using the `-Description` parameter?” The Help file states that the `-Description` parameter is a named parameter. This means you can use the `-Description` parameter only if you specify the parameter name, as shown earlier in this section. Following is the applicable portion of the Help file for the `-Description` parameter:

```
-Description <String[]>
  Gets only hotfixes with the specified descriptions. Wildcards are permitted. The default is all hotfixes on the computer.

  Required? false
  Position? named
  Default value All hotfixes
  Accept pipeline input? false
  Accept wildcard characters? True
```

### Finding specific processes

To find process information about a single process, I use the `-Name` parameter. Because the `-Name` parameter is the default (position 1) parameter for the *Get-Process* cmdlet, you do not have to specify the `-Name` parameter when calling *Get-Process* if you do not want to do so. For example, to find information about the Windows PowerShell process by using the *Get-Process* cmdlet, perform the following command at the command prompt of the Windows PowerShell console by using tab expansion:

```
Get-Pro + <TAB> + <SPACE> + Po + <TAB> + <ENTER>
```

The following example shows the command and associated output:

```
PS C:\> Get-Process powershell

Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName
------- ------ ----- ----- ----- ------ -- -----------
607 39 144552 164652 718 5.58 4860 powershell
```
You can tell the `Get-Process` cmdlet accepts the `-Name` parameter in a positional manner because the Help file states it is in position 1. The following example shows this position:

```
-Name <String[]>
  Specifies one or more processes by process name. You can type multiple process names (separated by commas) and use wildcard characters. The parameter name ("Name") is optional.

Required?    false
Position?    1
Default value
Accept pipeline input?    true (ByPropertyName)
Accept wildcard characters?  True
```

**NOTE** Be careful using positional parameters because they can be confusing. For example, the first parameter for the `Get-Process` cmdlet is the `–Name` parameter, but the first position parameter for the `Stop-Parameter` is the `–ID` parameter. As a best practice, always refer to the Help files to see what the parameters actually are called and the position in which they are expected. This is even more important when using a cmdlet with multiple parameters, such as the `Get-Random` cmdlet discussed in the following section.

**Generating random numbers in a range**

When used without any parameters, the `Get-Random` cmdlet returns a number that is in the range of 0 to 2,147,483,647. We have never had a Windows PowerShell user group meeting in which there were either 0 people in attendance or 2,147,483,647 people in attendance. Therefore, if you use the `Get-Random` cmdlet to select winners so you can hand out prizes at the end of the day, it is important to set a different minimum and maximum number.

**NOTE** When you use the `-Maximum` parameter for the `Get-Random` cmdlet, keep in mind that the maximum number never appears. Therefore, if you have 15 people attending your Windows PowerShell user group meeting, you should set the `-Maximum` parameter to 16 (unless you do not like the number 15 person and do not want him to win any prizes).

The default parameter for the `Get-Random` cmdlet is the `-Maximum` parameter. This means you can use the `Get-Random` cmdlet to generate a random number in the range of 0 to 20 by using tab expansion on the first line of the Windows PowerShell console. Remember that `Get-Random` never reaches the maximum number, so always use a number that is 1 greater than the desired upper number. Perform the following:

```
Get-R + <TAB> + <SPACE> + 21
```
If you want to generate a random number between 1 and 20, you might think you could use `Get-Random 1 21`, but that generates an error. The following example shows the command and error:

```
PS C:\> Get-Random 1 21
Get-Random : A positional parameter cannot be found that accepts argument '21'.
At line:1 char:1
+ Get-Random 1 21
+ ~~~~~~~~~~~~~~~~~~~
   + CategoryInfo : InvalidArgument: (:) [Get-Random], ParameterBindingException
```

The error states that a positional parameter cannot be found that accepts argument 21. This is because `Get-Random` has only one positional parameter, the `-Maximum` parameter. The `-Minimum` parameter is a named parameter. This parameter appears in the Help file for the `Get-Random` cmdlet. I show you how to use the Help files in Chapter 2, “Using Windows PowerShell cmdlets.”

To generate a random number in the range of 1 to 20, use named parameters. For assistance in creating the command, use tab expansion for the cmdlet name as well as for the parameter names. Perform the following at the command prompt to create the command using tab expansion:

```
Get-R + <TAB> + -M + <TAB> + <SPACE> + 21 + -M + <TAB> + <SPACE> + 1 + <ENTER>
```

The following example shows the command and associated output:

```
PS C:\> Get-Random -Maximum 21 -Minimum 1
19
```

**Introduction to parameter sets**

One potentially confusing characteristic of Windows PowerShell cmdlets is that there are often different ways of using the same cmdlet. For example, you can specify the `-Minimum` and `-Maximum` parameters, but you cannot also specify the `-Count` parameter. This is a bit unfortunate because it would seem that using the `-Minimum` and `-Maximum` parameters to specify the minimum and maximum numbers for random numbers makes sense. When the Windows PowerShell user group has five prizes to give away, it is inefficient to write a script to generate the five random numbers or run the same command five times.

This is where command sets come into play. The `-Minimum` and `-Maximum` parameters specify the range within which to pick a single random number. To generate more than one random number, use the `-Count` parameter. The following example shows the two parameter sets:

```
[<CommonParameters>]

[<CommonParameters>]
```
The first parameter set accepts -Maximum, -Minimum, and -SetSeed. The second parameter set accepts -InputObject, -Count, and -SetSeed. Therefore, you cannot use -Count with -Minimum or -Maximum because they are in two different groups of parameters (called parameter sets).

**NOTE** It is quite common for Windows PowerShell cmdlets to have multiple parameter sets. Tab expansion offers only parameters from one parameter set. Therefore, when you choose a parameter such as –Count from Get-Random, the non-compatible parameters do not appear in tab expansion. This feature keeps you from creating invalid commands. For an overview of cmdlets parameter sets, use the Get-Help cmdlet.

### Generating a certain number of random numbers

The Get-Random cmdlet, when used with the -Count parameter, accepts an -InputObject parameter. The -InputObject parameter is quite powerful. The following excerpt from the Help file states that it accepts a collection of objects:

```
-InputObject <Object[]>
` Specifies a collection of objects. Get-Random gets randomly selected objects in random order from the collection. Enter the objects, a variable that contains the objects, or a command or expression that gets the objects. You can also pipe a collection of objects to Get-Random.
```

An array (or a range) of numbers just happens to also be a collection of objects. The easiest way to generate a range (or an array) of numbers is to use the range operator. The range operator is two dots (periods) between two numbers. As shown in the following example, the range operator does not require spaces between the numbers and dots:

```
PS C:\> 1..5
1
2
3
4
5
```

Now, to pick five random numbers from the range of 1 to 10 requires only the command to appear here. The parentheses are required around the range of 1 to 10 numbers to ensure the range of numbers is created prior to selecting five from the collection:

```
Get-Random -InputObject (1..10) -Count 5
```
The following example shows the command and associated output:

```powershell
PS C:\> Get-Random -InputObject (1..10) -Count 5
7
5
10
1
8
```

**Using command-line utilities**

As easy as Windows PowerShell is to use, there are times when it is easier to find information by using a command-line utility. For example, to find IP configuration information, you need only use the `ipconfig.exe` utility. You can type this directly into the Windows PowerShell console and read the output in the console. The following example shows the command and associated output in truncated form:

```powershell
PS C:\> ipconfig

Windows IP Configuration

Wireless LAN adapter Local Area Connection* 14:
  Media State . . . . . . . . . . . : Media disconnected
  Connection-specific DNS Suffix . .

Ethernet adapter vEthernet (WirelessSwitch):
  Connection-specific DNS Suffix . : quadriga.com
  Link-local IPv6 Address . . . . . : fe80::915e:d324:aa0f:a54b%31
  IPv4 Address . . . . . . . . . . . : 192.168.13.220
  Subnet Mask . . . . . . . . . . . : 255.255.248.0
  Default Gateway . . . . . . . . . : 192.168.15.254

Wireless LAN adapter Local Area Connection* 12:
  Media State . . . . . . . . . . . : Media disconnected
  Connection-specific DNS Suffix . .

Ethernet adapter vEthernet (InternalSwitch):
  Connection-specific DNS Suffix . :
  Link-local IPv6 Address . . . . . : fe80::bd2d:5283:5572:5e77%19
  IPv4 Address . . . . . . . . . . . : 192.168.3.228
```
Subnet Mask : 255.255.255.0
Default Gateway : 192.168.3.100

To obtain the same information using Windows PowerShell, you need a more complex command. The command to obtain IP information is `Get-NetIPAddress`. But there are several advantages. For one thing, the output from the `IpConfig.exe` command is text, whereas the output from Windows PowerShell is an object. This means you can group, sort, filter, and format the output in an easy way.

The big benefit is that with the Windows PowerShell console, you have not only the simplicity of the command prompt, but you also have the powerful Windows PowerShell language built in. Therefore, if you need to refresh Group Policy three times and wait for five minutes between refreshes, you can use the command shown in the following example (looping is covered in Chapter 11, “Using Windows PowerShell Scripts”):

```
1..3 | % {gpupdate ; sleep 300}
```

---

**Working with Help options**

To use Help files effectively, the first thing you need to do is update them on your system. This is because Windows PowerShell 3.0 introduces a new model in which Help files update on a regular basis.

To update Help on your system, you must open the Windows PowerShell console with administrator rights. This is because Windows PowerShell Help files reside in the protected `Windows\System32\WindowsPowerShell` directory. Once you have launched the Windows PowerShell console with administrator rights, you need to ensure your computer has Internet access so it can download and install the updated files. If your computer does not have Internet connectivity, it will take several minutes before the command times out because Windows PowerShell tries really hard to obtain the updated files. If you run the `Update-Help` cmdlet with no parameters, Windows PowerShell attempts to download updated Help for all modules stored in the default Windows PowerShell modules locations that support updatable Help. To run `Update-Help` more than once a day, use the `-Force` parameter, as shown in the following example:

```
Update-Help -Force
```

Even without downloading updated Windows PowerShell Help, the Help subsystem displays the syntax of the cmdlet and other rudimentary information about the cmdlet.

To display Help information from the Internet, use the `-Online` switch. When used in this way, Windows PowerShell causes the default browser to open to the appropriate page from the Microsoft TechNet website.
In an enterprise, network administrators might want to use the *Save-Help* cmdlet to download Help from the Internet. Once downloaded, the *Update-Help* cmdlet can point to the network share for the files. This is an easy task to automate and can run as a scheduled task.

**Summary**

This chapter began with an overview of Windows PowerShell. In particular, it contrasted some of the differences and similarities between the Windows PowerShell console and the Windows PowerShell ISE. It explained that, regardless of where a Windows PowerShell command runs, the results are the same.

Windows PowerShell uses a verb and noun naming convention. To retrieve information, use the *Get* verb. To specify the type of information to obtain, use the appropriate noun. An example of this convention is the *Get-HotFix* cmdlet that returns hotfix information from the local system.

One of the most important concepts to understand about Windows PowerShell is that it allows a user to perform an action only if the security model permits it. For example, if a user has permission to stop a service by using the Services.MSC tool, the user will have permission to stop a service from within Windows PowerShell. But if a user is not permitted to stop a service elsewhere, Windows PowerShell does not permit the service to stop. Windows PowerShell also respects UAC. By default on Windows 7 and Windows 8, Windows PowerShell opens in least privilege mode. To perform actions requiring administrator rights, you must start Windows PowerShell as an administrator.

Many Windows PowerShell cmdlets run without any options and return valid data. This includes cmdlets such as *Get-Process* or *Get-Service*. However, most Windows PowerShell cmdlets require additional information to work properly. For example, the *Get-EventLog* cmdlet requires the name of a particular event log to return information.

The first thing you should do when logging onto the Windows PowerShell console is to run the *Update-Help* cmdlet. Note that this requires administrator rights and an Internet connection.
When you need to use Windows PowerShell on your local computer, it is pretty easy: You open the Windows PowerShell console or the Windows PowerShell ISE, and you run a command or a series of commands. Assuming you have rights to make the changes in the first place, it just works. But what if the change you need to make must be enacted on a hundred or a thousand computers? In the past, these types of changes required expensive specialized software packages, but with Windows PowerShell 3.0 running a command on a remote computer is as easy as running the command on your local computer; in some cases, it is even easier.

Using Windows PowerShell remoting

One of the great improvements in Windows PowerShell 3.0 is the change surrounding remoting. The configuration is easier than it was in Windows PowerShell 2.0, and in many cases, Windows PowerShell remoting “just works.” When we talk about Windows PowerShell remoting, a bit of confusion can arise because there are several different ways of running commands against remote servers. Depending on your particular network configuration and security needs, one or more methods of remoting might not be appropriate.

Classic remoting

Classic remoting relies on protocols such as the Distributed Component Object Model (DCOM) and remote procedure call (RPC) to make connections to remote machines. Traditionally, these techniques require opening many ports in the firewall and starting various services the different cmdlets utilize. To find the Windows PowerShell cmdlets that natively support remoting, use the *Get-Help* cmdlet. Specify a value of computername for the parameter of the *Get-Help* cmdlet. This command produces a nice list of all cmdlets that have native support for remoting. The following example shows the command and associated
output (this command does not display all cmdlets with support for computername unless the
associated modules are preloaded):

PS C:\> Get-Help * -Parameter computername | sort name | ft name, synopsis -auto -wrap

<table>
<thead>
<tr>
<th>Name</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add-Computer</td>
<td>Add the local computer to a domain or workgroup.</td>
</tr>
<tr>
<td>Add-Printer</td>
<td>Adds a printer to the specified computer.</td>
</tr>
<tr>
<td>Add-PrinterDriver</td>
<td>Installs a printer driver on the specified computer.</td>
</tr>
<tr>
<td>Add-PrinterPort</td>
<td>Installs a printer port on the specified computer.</td>
</tr>
</tbody>
</table>

Some of the cmdlets provide the ability to specify credentials. This allows you to use a dif-
ferent user account to make the connection and retrieve the data.

The following example shows this technique of using the computername and the creden-
tial parameters in a cmdlet:

PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName ex1 -Credential
nwtraders\administrator

<table>
<thead>
<tr>
<th>TimeCreated</th>
<th>ProviderName</th>
<th>Id</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>7/1/2012</td>
<td>MSExchange ADAccess</td>
<td>2080</td>
<td>Process MAD.EXE (...</td>
</tr>
</tbody>
</table>

However, as mentioned earlier, use of these cmdlets often requires opening holes in the
firewall or starting specific services. By default, these types of cmdlets fail when run against
remote machines that have not relaxed access rules. The following example shows this type of
error:

PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential
nwtraders\administrator
Get-WinEvent : The RPC server is unavailable
At line 1:
  Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential iam

Other cmdlets, such as Get-Service or Get-Process, do not have a credential parameter, and
therefore the command impersonates the logged-on user, as shown in the following example:

PS C:\> Get-Service -ComputerName hyperv -Name bits

<table>
<thead>
<tr>
<th>Status</th>
<th>Name</th>
<th>DisplayName</th>
</tr>
</thead>
</table>
| Running| bits   | Background Intelligent Transfer Ser...            

PS C:\>
Just because the cmdlet does not support alternative credentials does not mean the cmdlet must impersonate the logged-on user. Holding down the Shift key and right-clicking on the Windows PowerShell icon brings up an action menu that allows you to run the program as a different user. When you use the Run as different user dialog box, you have alternative credentials available for Windows PowerShell cmdlets that do not support the credential parameter.

Configuring Windows PowerShell remoting

Windows Server 2012 installs with Windows Remote Management (WinRm) configured and running to support remote Windows PowerShell commands. WinRm is the Microsoft implementation of the industry standard WS-Management Protocol. As such, WinRM provides a firewall-friendly method of accessing remote systems in an interoperable manner. It is the remoting mechanism used by the new Common Information Model (CIM) cmdlets (the CIM cmdlets are covered in Chapter 9, “Using CIM”). As soon as Windows Server 2012 is up and running, you can make a remote connection and run commands or open an interactive Windows PowerShell console. A Windows 8 client, on the other hand, ships with WinRm locked down. Therefore, the first step is to use the `Enable-PSRemoting` function to configure remoting. When running the `Enable-PSRemoting` function, the following steps occur:

1. Starts or restarts the WinRM service.
2. Sets the WinRM service startup type to Automatic.
3. Creates a listener to accept requests from any Internet Protocol (IP) address.
4. Enables inbound firewall exceptions for `WS_Management` traffic.
5. Sets a target listener named `Microsoft.powershell`.
7. Sets a target listener named `Microsoft.powershell32`.

   During each step of this process, the function prompts you to agree or not agree to performing the specified action. If you are familiar with the steps the function performs, and you do not make any changes from the defaults, you can run the command with the Force switched parameter and it will not prompt prior to making the changes. The following example shows the syntax of this command:

```
Enable-PSRemoting -Force
```
The following example shows the use of the `Enable-PSRemoting` function in interactive mode, along with all associated output from the command:

```
PS C:\> Enable-PSRemoting

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable remote management of this computer by using the Windows Remote Management (WinRM) service.
This includes:
1. Starting or restarting (if already started) the WinRM service
2. Setting the WinRM service startup type to Automatic
3. Creating a listener to accept requests on any IP address
4. Enabling Windows Firewall inbound rule exceptions for WS-Management traffic (for http only).

Do you want to continue?
(default is "Y"): y
WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.

WinRM has been updated for remote management.
Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP on this machine.
WinRM firewall exception enabled.

Confirm
Are you sure you want to perform this action?
(default is "Y"): y

Confirm
Are you sure you want to perform this action?
(default is "Y"): y

Confirm
Are you sure you want to perform this action?
(default is "Y"): y
```

PS C:\>
Once configured, use the `Test-WSMan` cmdlet to ensure the WinRM remoting is properly configured and is accepting requests. A properly configured system replies with the following data:

```
PS C:\> Test-WSMan -ComputerName w8c504
```

```
wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

This cmdlet works with Windows PowerShell 2.0 remoting as well. Keep in mind that configuring WinRM through the `Enable-PSRemoting` function does not enable the WinRM firewall exception, and therefore PING commands will not work by default when pinging to a Windows 8 client system.

**Running commands**

For simple configuration on a single remote machine, entering a remote Windows PowerShell session is the answer. To enter a remote Windows PowerShell session, use the `Enter-PSSession` cmdlet to create an interactive remote Windows PowerShell session on a target machine. If you do not supply credentials, the remote session impersonates your current logon. The output in the following example illustrates connecting to a remote computer named dc1:

```
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> sl C:\
[dc1]: PS C:\> gwmi win32_bios
```

```
SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
```

```
[dc1]: PS C:\> exit
PS C:\>
```

Once established, the Windows PowerShell prompt changes to include the name of the remote system. The `Set-Location` (`sl` is an alias) changes the working directory on the remote system to `C:\`. Next, the `Get-WmiObject` cmdlet retrieves the BIOS information on the remote system. The `Exit` command exits the remote session and the Windows PowerShell prompt returns to the default.

The good thing is that when using the Windows PowerShell transcript tool through `Start-Transcript`, the transcript tool captures output from the remote Windows PowerShell session as well as output from the local session. Indeed, all commands typed appear in the transcript.
The following commands illustrate beginning a transcript, entering a remote Windows PowerShell session, typing a command, exiting the session, and stopping the transcript:

```
PS C:\> Start-Transcript
Transcript started, output file is C:\Users\administrator.IAMMRED\Documents\PowerShell_transcript.20120701124414.txt
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> gwmi win32_bios

SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6

[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Stop-Transcript
Transcript stopped, output file is C:\Users\administrator.IAMMRED\Documents\PowerShell_transcript.20120701124414.txt
PS C:\>
```

Figure 7-1 shows the transcript from the preceding remote Windows PowerShell session. The transcript contains all commands, including the ones from the remote computer, and associated output.

---

**Figure 7-1** The transcript tool works in remote Windows PowerShell sessions as well as in local Windows PowerShell console sessions.
Running a single Windows PowerShell command

When you have a single command to run, it does not make sense to go through all the trouble of building and entering an interactive, remote Windows PowerShell session. Instead of creating a remote Windows PowerShell console session, you can run a single command by using the *Invoke-Command* cmdlet. If you have a single command to run, use the cmdlet directly and specify the computer name as well as any credentials required for the connection. The following example shows this technique, with the last process running on the ex1 remote server:

```
PS C:\> Invoke-Command -ComputerName ex1 -ScriptBlock {gps | select -Last 1}
```

<table>
<thead>
<tr>
<th>Handles</th>
<th>NPM(K)</th>
<th>PM(K)</th>
<th>WS(K)</th>
<th>VM(M)</th>
<th>CPU(s)</th>
<th>Id</th>
<th>ProcessName</th>
<th>PSComputerName</th>
</tr>
</thead>
<tbody>
<tr>
<td>224</td>
<td>34</td>
<td>47164</td>
<td>51080</td>
<td>532</td>
<td>0.58</td>
<td>10164</td>
<td>wsmprovhost</td>
<td>ex1</td>
</tr>
</tbody>
</table>

When you work interactively in a Windows PowerShell console, you might not want to type a long command, even when using tab expansion to complete the command. To shorten the amount of typing, you can use the `icm` alias for the *Invoke-Command* cmdlet. You can also rely upon positional parameters (the first parameter is the computer name and the second parameter is the script block). By using aliases and positional parameters, the previous command shortens considerably, as shown in the following example:

```
PS C:\> icm ex1 {gps | select -l 1}
```

<table>
<thead>
<tr>
<th>Handles</th>
<th>NPM(K)</th>
<th>PM(K)</th>
<th>WS(K)</th>
<th>VM(M)</th>
<th>CPU(s)</th>
<th>Id</th>
<th>ProcessName</th>
<th>PSComputerName</th>
</tr>
</thead>
<tbody>
<tr>
<td>221</td>
<td>34</td>
<td>47260</td>
<td>51048</td>
<td>536</td>
<td>0.33</td>
<td>4860</td>
<td>wsmprovhost</td>
<td>ex1</td>
</tr>
</tbody>
</table>

Running a single command against multiple computers

Use of the *Invoke-Command* exposes one of the more powerful aspects of Windows PowerShell remoting, which is running the same command against a large number of remote systems. The secret behind this power is that the computername parameter from the *Invoke-Command* cmdlet accepts an array of computer names. In the output appearing here, an array of computer names is stored in the variable `$cn`. Next, the `$cred` variable holds the *credential* object for the remote connections. Finally, the *Invoke-Command* cmdlet is used to make connections to all the remote machines and to return the BIOS information from the systems. The nice thing about this technique is that an additional parameter, PSComputerName, is added to the returning object, permitting easy identification of which BIOS is associated with which computer system. The following example shows the commands and associated output:

```
PS C:\> $cn = "dc1","dc3","ex1","sql1","wsus1","wds1","hyperv1","hyperv2","hyperv3"
PS C:\> $cred = Get-Credential iammred\administrator
PS C:\> Invoke-Command -cn $cn -cred $cred -ScriptBlock {gwmi win32_bios}
```
<table>
<thead>
<tr>
<th>SMBIOSBIOSVersion</th>
<th>Manufacturer</th>
<th>Name</th>
<th>SerialNumber</th>
<th>Version</th>
<th>PSComputerName</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAP6710H.86A.0072.2011.0927.1425</td>
<td>Intel Corp.</td>
<td>BIOS Date: 09/27/11 14:25:42 Ver: 04.06.04</td>
<td></td>
<td>INTEL - 1072009</td>
<td>hyperv3</td>
</tr>
<tr>
<td>A11</td>
<td>Dell Inc.</td>
<td>Phoenix ROM BIOS PLUS Version 1.10 A11</td>
<td>BDY91L1</td>
<td>DELL - 15</td>
<td>hyperv2</td>
</tr>
<tr>
<td></td>
<td>Dell Computer Corporation</td>
<td>Default System BIOS</td>
<td>9HQ1S21</td>
<td>DELL - 6</td>
<td>dc1</td>
</tr>
<tr>
<td>090004</td>
<td>American Megatrends Inc.</td>
<td>BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04</td>
<td>3692-0963-1044-7503-9631-2546-83</td>
<td>VIRTUAL - 3000919</td>
<td>wsus1</td>
</tr>
<tr>
<td>V1.6</td>
<td>American Megatrends Inc.</td>
<td>Default System BIOS</td>
<td></td>
<td>7583MS - 20091228</td>
<td>hyperv1</td>
</tr>
<tr>
<td>080015</td>
<td>American Megatrends Inc.</td>
<td>Default System BIOS</td>
<td>None</td>
<td>091709 - 20090917</td>
<td>sql1</td>
</tr>
<tr>
<td>080015</td>
<td>American Megatrends Inc.</td>
<td>Default System BIOS</td>
<td>None</td>
<td>091709 - 20090917</td>
<td>wds1</td>
</tr>
<tr>
<td>090004</td>
<td>American Megatrends Inc.</td>
<td>BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04</td>
<td>8994-9999-0865-2542-2186-8044-69</td>
<td>VIRTUAL - 3000919</td>
<td>dc3</td>
</tr>
</tbody>
</table>
Creating a persisted connection

If you anticipate making multiple connections to a remote system, use the `New-PSSession` cmdlet to create a remote Windows PowerShell session. The `New-PSSession` cmdlet permits you to store the remote session in a variable and provides you with the ability to enter and leave the remote session as often as required, without the additional overhead of creating and destroying remote sessions. In the commands that follow, a new Windows PowerShell session is created through the `New-PSSession` cmdlet. The newly created session is stored in the `$dc1` variable. Next, the `Enter-PSSession` cmdlet is used to enter the remote session by using the stored session. A command retrieves the remote hostname, and the remote session is exited through the `Exit` command. Next, the session is re-entered and the last process retrieved. The session is exited once again. Finally, the `Get-PSSession` cmdlet retrieves Windows PowerShell sessions on the system, and all sessions are removed through the `Remove-PSSession` cmdlet:

```
PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential iammred\administrator
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> hostname
dc1
[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> gps | select -Last 1
 Handles  NPM(K)   PM(K)   WS(K) VM(M)   CPU(s)   Id ProcessName
------- ------ ------ ----- ----- ------ ------  ----------
   292     9  39536  50412  158  1.97   2332 wsmprovhost

[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Get-PSSession

   Id Name            ComputerName    State         ConfigurationName     Availability
   -- ----            ------------    -----         -----------------     ------------
     8 Session8        dc1             Opened        Microsoft.PowerShell     Available

PS C:\> Get-PSSession | Remove-PSSession
PS C:\>
```
If you have several commands, or if you anticipate making multiple connections, the `Invoke-Command` cmdlet accepts a session parameter in the same manner as the `Enter-PSSession` cmdlet does. In the output appearing here, a new `PSSession` is created to a remote computer named `dc1`. The remote session is used to retrieve two different pieces of information. Once completed, the session stored in the `$dc1` variable is explicitly removed:

```powershell
PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential iammred\administrator
PS C:\> Invoke-Command -Session $dc1 -ScriptBlock {hostname}
dc1
PS C:\> Invoke-Command -Session $dc1 -ScriptBlock {Get-EventLog application -Newest 1}
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Time</th>
<th>EntryType</th>
<th>Source</th>
<th>InstanceID</th>
<th>Message</th>
<th>PSComputerName</th>
</tr>
</thead>
<tbody>
<tr>
<td>17702</td>
<td>Jul 01 12:59</td>
<td>Information</td>
<td>ESENT</td>
<td>701</td>
<td>DFSR...</td>
<td>dc1</td>
</tr>
</tbody>
</table>

`PS C:\> Remove-PSSession $dc1`

You can also create persisted connection to multiple computers. This enables you to use the `Invoke-Command` cmdlet to run multiple commands against multiple remote computers. The first thing is to create a new `PSSession` that contains multiple computers. You can do this by using alternative credentials. Create a variable that holds the credential object returned by the `Get-Credential` cmdlet. A dialog box appears, permitting you to enter the credentials. Figure 7-2 shows the dialog box.

![Windows PowerShell Credential...](image)

**FIGURE 7-2** Store remote credentials in a variable populated through the `Get-Credential` cmdlet.

Once you have stored the credentials in a variable, create another variable to store the remote computer names. Next, use the `New-PSSession` cmdlet to create a new Windows PowerShell session using the computer names stored in the computer name variable and the credentials stored in the credential variable. To be able to reuse the Windows PowerShell remote session, store the newly created Windows PowerShell session in a variable as well. The following example illustrates storing the credentials, computer names, and newly created Windows PowerShell session:
$cred = Get-Credential -Credential iammred\administrator
$cn = "ex1","dc3"
$ps = New-PSSession -ComputerName $cn -Credential $cred

Once the Windows PowerShell session is created and stored in a variable, it can be used to execute commands against the remote computers. To do this, use the *Invoke-Command* cmdlet, as shown in the following example:

```powershell
PS C:\> Invoke-Command -Session $ps -ScriptBlock {gsv | select -First 1}
```

<table>
<thead>
<tr>
<th>Status</th>
<th>Name</th>
<th>DisplayName</th>
<th>PSComputerName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stopped</td>
<td>AeLookupSvc</td>
<td>Application Experience</td>
<td>ex1</td>
</tr>
<tr>
<td>Running</td>
<td>ADWS</td>
<td>Active Directory Web Services</td>
<td>dc3</td>
</tr>
</tbody>
</table>

The great thing about storing the remote connection in a variable is that it can be used for additional commands as well. The following example shows the command that returns the first process from each of the two remote computers:

```powershell
PS C:\> Invoke-Command -Session $ps -ScriptBlock {gps | select -First 1}
```

<table>
<thead>
<tr>
<th>Handles</th>
<th>NPM(K)</th>
<th>PM(K)</th>
<th>WS(K)</th>
<th>VM(M)</th>
<th>CPU(s)</th>
<th>Id</th>
<th>ProcessName</th>
<th>PSComputerName</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>7</td>
<td>1812</td>
<td>6980</td>
<td>53</td>
<td>0.70</td>
<td>3300 conhost</td>
<td>dc3</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>824</td>
<td>2520</td>
<td>22</td>
<td>0.22</td>
<td>1140 conhost</td>
<td>ex1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7-3 shows the commands to store the credentials, create a remote Windows PowerShell connection to two different computers, and run two remote commands against them. Figure 7-3 also shows the output associated with the commands.

**FIGURE 7-3** By creating and by storing a remote Windows PowerShell connection, it becomes easy to run commands against multiple computers.
Troubleshooting Windows PowerShell remoting

The first tool to use to see if Windows PowerShell remoting is working or not is the Test-WSMan cmdlet. Use it first on the local computer (no parameters are required). The following example shows the command and associated output:

PS C:\> Test-WSMan

wsnid : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProductVendor : Microsoft Corporation
ProductVersion : OS: 0.0.0 SP: 0.0 Stack: 3.0

To test a remote computer, specify the -ComputerName parameter. The following example shows the command running against a Windows Server 2012 domain controller named dc3:

PS C:\> Test-WSMan -ComputerName dc3

wsnid : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProductVendor : Microsoft Corporation
ProductVersion : OS: 0.0.0 SP: 0.0 Stack: 3.0

However, the Test-WSMan cmdlet also works against a computer running Windows PowerShell 2.0. The following example shows the command running against a Windows Server 2008 domain controller named dc1:

PS C:\> Test-WSMan -ComputerName dc1

wsnid : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProductVendor : Microsoft Corporation
ProductVersion : OS: 0.0.0 SP: 0.0 Stack: 2.0

To examine a specific Windows PowerShell session, use the Get-PSSession cmdlet. The easiest way to do this is to pipeline the variable containing the Windows PowerShell session to the Get-PSSession cmdlet. The key items to pay attention to are the computer name, the state of the session, and the availability of the session. The following example shows this technique:

PS C:\> $ps | Get-PSSession

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>ComputerName</th>
<th>State</th>
<th>ConfigurationName</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Session3</td>
<td>ex1</td>
<td>Opened</td>
<td>Microsoft.PowerShell</td>
<td>Available</td>
</tr>
<tr>
<td>4</td>
<td>Session4</td>
<td>dc3</td>
<td>Opened</td>
<td>Microsoft.PowerShell</td>
<td>Available</td>
</tr>
</tbody>
</table>
To focus on a specific session, reference the session by either ID or by Name. Send the returned session object over the pipeline to the `Format-List` cmdlet and select all the properties. The following example shows this technique (using `fl` as an alias for the `Format-List` cmdlet):

```powershell
PS C:\> Get-PSSession -Name Session4 | fl *
```

<table>
<thead>
<tr>
<th>State</th>
<th>Opened</th>
</tr>
</thead>
<tbody>
<tr>
<td>IdleTimeout</td>
<td>72000000</td>
</tr>
<tr>
<td>OutputBufferingMode</td>
<td>None</td>
</tr>
<tr>
<td>ComputerName</td>
<td>dc3</td>
</tr>
<tr>
<td>ConfigurationName</td>
<td>Microsoft.PowerShell</td>
</tr>
<tr>
<td>InstanceId</td>
<td>c15cc80d-64f0-4096-a010-0211f0188aec</td>
</tr>
<tr>
<td>Id</td>
<td>4</td>
</tr>
<tr>
<td>Name</td>
<td>Session4</td>
</tr>
<tr>
<td>Availability</td>
<td>Available</td>
</tr>
<tr>
<td>ApplicationPrivateData</td>
<td>{PSVersionTable}</td>
</tr>
</tbody>
</table>

You can remove a remote Windows PowerShell session by pipelining the results of `Get-PSSession` to the `Remove-PSSession` cmdlet, as shown in the following example:

```powershell
Get-PSSession -Name Session4 | Remove-PSSession
```

You can also remove a PS session directly by specifying the name to the `Remove-PSSession` cmdlet, as shown in the following example:

```powershell
Remove-PSSession -Name session3
```

**Summary**

This chapter discussed the reason to use Windows PowerShell remoting. We covered the different types of remoting, such as classic remoting and Windows Remote Management Windows PowerShell (WinRM) remoting. In addition, we covered how to enable Windows PowerShell remoting and how to run a single command against a remote computer. Finally, we examined running multiple commands, creating persisted connections, and troubleshooting Windows PowerShell remoting.
Index

A
About conceptual Help topics, 29–30
accessing PowerShell, Start window, 2–4
AccountsWithNoRequiredPassword.ps1 script, 154
Action parameter, 209
Add-Content cmdlet, 87
Add Criteria button, 66
AddTwo function, 211
administrative rights
  error handling, 225–226
  security issues, 5–6
aliases
  data type, functions, 190
  properties, Get-Process cmdlet, 54
Alias provider, 80–82
AllSigned level (script support), 36, 156
appending information, text files, 70–71
arguments, positional, 187
arrays
  evaluating with Switch statement, 181
  indexing, 138
$_ automatic variable, 159
AutoSize parameter, 56, 61–62

B
backing up profiles, 38
backtick (`) character, 158
BadScript.ps1 script, 206
breakpoints, debugging scripts
deleting, 215–216
listing, 213–215
responding to, 211–212
setting, 204–211
commands, 209–211
line number, 204–205
variables, 206–209
Break statement, scripts, 174
BusinessLogicDemo.ps1 script, 194
business logic, encapsulating with functions, 194–196
buttons, Add Criteria, 66
Bypass level (script support), 36, 156

C
$_ automatic variable, 159
ASCII values, DemoDoWhile.ps1 script, 167
assignment operators, 177–178
associations
  CIM (Common Information Model), 134–140
  qualifiers, 131
asterisk (*) parameter, 82
attempt and fail, handling missing rights, 226
$caption variable, 220
Case Else expression, 179
case-sensitivity issues, 10
Certificate provider, 82–84
checking for rights, 226
choice limitations, error handling, 220–225
  contains operator, 223–225
  PromptForChoice method, 220–221
  Test-Connection cmdlet, 222
$choiceRTN variable, 221
CIM (Common Information Model), 127–140
  associations, 134–140
  cleaning up output, 134
<table>
<thead>
<tr>
<th>cmdlets, 127–132</th>
</tr>
</thead>
<tbody>
<tr>
<td>classname parameter, 128</td>
</tr>
<tr>
<td>filtering WMI classes by qualifiers, 130–132</td>
</tr>
<tr>
<td>finding WMI class methods, 128–129</td>
</tr>
<tr>
<td>reducing returned properties and instances, 133</td>
</tr>
<tr>
<td>retrieving WMI instances, 132</td>
</tr>
<tr>
<td>classes</td>
</tr>
<tr>
<td>Win32_PingStatus WMI, 222</td>
</tr>
<tr>
<td>WMI (Windows Management Instrumentation), 115–117, 127–132</td>
</tr>
<tr>
<td>classic remoting, 99–101</td>
</tr>
<tr>
<td>class methods (WMI), 128–129</td>
</tr>
<tr>
<td>class name parameter, 128</td>
</tr>
<tr>
<td>cmdlets, 6–16</td>
</tr>
<tr>
<td>About conceptual Help topics, 29–30</td>
</tr>
<tr>
<td>Add-Content, 87</td>
</tr>
<tr>
<td>CIM (Common Information Model), 127–132</td>
</tr>
<tr>
<td>classname parameter, 128</td>
</tr>
<tr>
<td>filtering WMI classes by qualifiers, 130–132</td>
</tr>
<tr>
<td>finding WMI class methods, 128–129</td>
</tr>
<tr>
<td>debugging scripts, 203–204</td>
</tr>
<tr>
<td>Disable-PSBreakpoint, 215</td>
</tr>
<tr>
<td>Enable-NetAdapter, 42</td>
</tr>
<tr>
<td>Enable-PSBreakpoint, 215</td>
</tr>
<tr>
<td>Enter-PSSession, 103, 107</td>
</tr>
<tr>
<td>Export-Clixml, 77</td>
</tr>
<tr>
<td>Export-CSV, 73–76</td>
</tr>
<tr>
<td>ForEach-Object, 159, 173</td>
</tr>
<tr>
<td>Format-List, 54, 58–61, 111</td>
</tr>
<tr>
<td>Format-Table, 53–57</td>
</tr>
<tr>
<td>controlling table display, 55–57</td>
</tr>
<tr>
<td>Format-Wide, 61–63</td>
</tr>
<tr>
<td>Get-Alias, 165</td>
</tr>
<tr>
<td>Get-AppxPackage, 50–51</td>
</tr>
<tr>
<td>Get-ChildItem, 80, 88, 90</td>
</tr>
<tr>
<td>Get-CimAssociatedInstance, 134, 138</td>
</tr>
<tr>
<td>Get-CimClass, 127</td>
</tr>
<tr>
<td>Get-CimInstance, 132</td>
</tr>
<tr>
<td>Get-Command, 30–33, 183–185</td>
</tr>
<tr>
<td>Get-Content, 87</td>
</tr>
<tr>
<td>Get-Credential, 108</td>
</tr>
<tr>
<td>Get-Culture, 10–11</td>
</tr>
<tr>
<td>Get-Date, 11</td>
</tr>
<tr>
<td>Get-EventLog, 49–50</td>
</tr>
<tr>
<td>Get-Help, 26–28</td>
</tr>
<tr>
<td>Get-Hotfix, 8</td>
</tr>
<tr>
<td>Get-ISESnippet, 150</td>
</tr>
<tr>
<td>Get-ItemProperty, 90</td>
</tr>
<tr>
<td>Get-Member, 33–34</td>
</tr>
<tr>
<td>Get-NetAdapter, 9, 42</td>
</tr>
<tr>
<td>Get-NetConnectionProfile, 10</td>
</tr>
<tr>
<td>Get-Process, 7–8, 42</td>
</tr>
<tr>
<td>Get-PSDrive, 89</td>
</tr>
<tr>
<td>Get-PSProvider, 80</td>
</tr>
<tr>
<td>Get-Random, 12</td>
</tr>
<tr>
<td>Get-Service, 9, 43</td>
</tr>
<tr>
<td>Get-UCulture, 11</td>
</tr>
<tr>
<td>Get-VM, 42</td>
</tr>
<tr>
<td>Get-WinEvent, 27–28</td>
</tr>
<tr>
<td>Get-WmiObject, 114</td>
</tr>
<tr>
<td>Group-Object, 44–46</td>
</tr>
<tr>
<td>Help topics, 26</td>
</tr>
<tr>
<td>Import-Clixml, 77</td>
</tr>
<tr>
<td>Import-Module, 83</td>
</tr>
<tr>
<td>intellisense display, 147–148</td>
</tr>
<tr>
<td>Invoke-Command, 105</td>
</tr>
<tr>
<td>Invoke-Item, 83</td>
</tr>
<tr>
<td>New-Alias, 82</td>
</tr>
<tr>
<td>New-IseSnippet, 149</td>
</tr>
<tr>
<td>New-Item, 38, 94</td>
</tr>
<tr>
<td>New-PSDrive, 89</td>
</tr>
<tr>
<td>New-PSSession, 107</td>
</tr>
<tr>
<td>New-WebServiceProxy, 191</td>
</tr>
<tr>
<td>Out-File, 72</td>
</tr>
<tr>
<td>Out-GridView, 63–67</td>
</tr>
<tr>
<td>parameters, 22–24</td>
</tr>
<tr>
<td>-entryType, 50</td>
</tr>
<tr>
<td>ErrorAction parameter, 23–24</td>
</tr>
<tr>
<td>Verbose, 22–23</td>
</tr>
<tr>
<td>parameter sets, 16–18</td>
</tr>
<tr>
<td>Pop-Location, 92, 94</td>
</tr>
<tr>
<td>property members, 34–35</td>
</tr>
<tr>
<td>Push-Location, 92</td>
</tr>
<tr>
<td>Read-Host, 186</td>
</tr>
<tr>
<td>Remove-Item, 81, 150</td>
</tr>
<tr>
<td>Remove-PSBreakpoint, 215</td>
</tr>
<tr>
<td>Remove-PSSession, 107</td>
</tr>
<tr>
<td>Save-Help, 20</td>
</tr>
<tr>
<td>Select-Object, 115</td>
</tr>
<tr>
<td>Select-String, 118–120</td>
</tr>
<tr>
<td>Set-ExecutionPolicy, 36–38, 156–157, 226</td>
</tr>
<tr>
<td>Set-Item, 93</td>
</tr>
<tr>
<td>Set-ItemProperty, 94</td>
</tr>
<tr>
<td>Set-Location, 81, 87, 92</td>
</tr>
<tr>
<td>Set-PropertyItem, 94</td>
</tr>
<tr>
<td>Set-PSBreakpoint, 204</td>
</tr>
</tbody>
</table>
Set-PSDebug, 203
Show-Command, 34–35
single parameters, 12–16
  Description parameter, 13–14
  Maximum parameter, 15–16
  Name parameter, 14–15
Sort-Object, 42, 47–48, 76
Start-Transcript, 24, 38
Start-VM, 42
Stop-Transcript, 25
Test-Connection, 222
Test-Path, 38, 92, 95
Test-WSMan, 103, 110
two-part name, 6
Update-Help, Force parameter, 19–20
Win32_Process, 135
Write-Host, 209
$cn variable, 105
code, creating with ISE snippets, 148–149
column buttons, output grids, 64–66
Column parameter, 63
Command Add-on, Script pane, 145
Command Add-on window, 142–143
command-line parameters, assigning default values to, 218–219
command-line utilities, 18–19
Command parameter, 209–211
commands. See cmdlets
debugger commands, 211
remoting, 103–107
Script pane, 145
  setting breakpoints on, 209–211
Comma Separated Value files. See .csv files
common classes, WMI, 115
Common Information Model. See CIM
cmparison operators, 176–178
complex objects, storing in XML, 76–78
computername parameter, 105
concatenation operator, 159
conceptual Help topics, 26
conditions, If statements, 178–179
configuring remoting, 101–103
Confirm parameter (cmdlets), 22
console, 1
constants, scripts, 160–161
construction, While statement, 162–164
consumers, WMI (Windows Management Instrumentation), 113
contains operator, 223–225
core classes, WMI, 115
creating
code, ISE snippets, 148–149
default value for missing parameters, 218–220
For...Loop, 170–172
functions, 185
lists, 58–61
  properties by name, 59
  properties by wildcard, 59–61
output grids, 63–67
  column buttons, 64–66
  filter box, 66–67
profiles, 37–38
registry keys, 92–93
snippets (ISE), 149–150
tables, 53–57
  display, 55–57
  ordering properties, 54–55
$cred variable, 105
.csv (Comma Separated Value) files, storing output, 73–76
  NoTypeInformation parameter, 73–75
type information, 75–76
culture settings, Get-Culture cmdlet, 10–11
CurrentUser scope (execution policy), 36, 156
customizing Format-Wide output, 62–63

D
data
  leveraging providers. See providers
storage. See storing output
data pipeline, 41
  filtering output, 46–51
  grouping output after sorting, 44–46
  sorting output from a cmdlet, 42–44
datasets, 117
data type aliases, functions, 190
dates, retrieving with Get-Date cmdlet, 11
DateTime object, 48
DCOM (Distributed Component Object Model), 99
debugging scripts, 203–216
cmdlets, 203–204
deleting breakpoints, 215–216
listing breakpoints, 213–215
Debug parameter (cmdlets)

responding to breakpoints, 211–212
setting breakpoints, 204–211
commands, 209–211
line number, 204–205
variables, 206–209
Debug parameter (cmdlets), 22
declared variables, 160
DefaultDisplayPropertySet configuration, 119
Default statement, scripts, 180
default values, missing parameters, 218–220
deleting breakpoints, 215–216
DemoBreakFor.ps1 script, 174
DemoDoWhile.ps1 script, 166–168
DemoDoWhile.vbs script, 165
DemoExitFor.ps1 script, 175
DemoExitFor.vbs script, 174
DemoForEachNext.vbs script, 172
DemoForEach.ps1 script, 173
DemoForLoop.ps1 script, 170
DemoForLoop.vbs script, 170
DemoForWithoutInitOrRepeat.ps1 script, 170
demolfElseIfElse.ps1 script, 178
demolfElseIfElse.vbs script, 178
Demos.ps1 script, 176
Demos.vbs script, 176
DemoSelectCase.vbs script, 179
DemoSwitchArrayBreak.ps1 script, 182
DemoSwitchArray.ps1 script, 181
DemoSwitchCase.ps1 script, 180
DemoSwitchMultiMatch.ps1 script, 181
DemoTrapSystemException.ps1 script, 191–192
DemoWhileLessThan.ps1 script, 162
Deployment Image Servicing and Management (DISM)
Get-WindowsDriver function, 47
deprecated qualifier, 130
Description parameter (cmdlets), 13–14
Description parameter, snippets, 149
dialog boxes, UAC (User Account Control), 5
DirectoryListWithArguments.ps1 script, 154
Disable-PSBreakpoint cmdlet, 215
DISM (Deployment Image Servicing and Management)
Get-WindowsDriver function, 47
display
formatting output, 61–63
lists, 61
tables, 55–57
DisplayCapitalLetters.ps1 script, 167
Distributed Component Object Model (DCOM), 99
DivideNum function, 211
Do...Loop statement, scripts, 166–170
DotSourceScripts.ps1 script, 198
Do...Until statement, scripts, 168
DoWhileAlwaysRuns.ps1 script, 169–170
Do...While statements, scripts, 165–168
  casting to ASCII values, 167–168
  operating over an array, 166–167
  range operators, 166
drives, Registry provider, 89–90
dynamic classes, WMI, 115
dynamic qualifier, 131

ease of modification, functions, 196–201
editing profiles, 38
Enable-Netadapter cmdlet, 42
Enable-PSBreakpoint cmdlet, 215
Enable-PSRemoting function, 101
  enabling support, scripts, 156–157
encapsulating business logic, functions, 194–196
EndlessDoUntil.ps1 script, 169
Enter-PSSession cmdlet, 103, 107
EntryType parameter, 50
Environment provider, 85–86
ErrorAction parameter, 158
ErrorAction parameter (cmdlets), 23–24
error handling
  forcing intentional errors, 24
  limiting choices, 220–225
    contains operator, 223–225
    PromptForChoice method, 220–221
    Test-Connection cmdlet, 222
  missing parameters, 217–220
    creating default value, 218–220
    making parameters mandatory, 219–220
  missing rights, 225–226
  Try/Catch/Finally block, 227–228
ErrorVariable parameter (cmdlets), 22
escape character, 163
Examples parameter, Get-Help cmdlet, 27
Exit For statements (VBScript), 174
exiting ForEach statement, 174–176
Exit statement, scripts, 174–175
expanding strings, 163
expired certificates, 84–85
ExpiringInDays parameter, 84
Export-Clixml cmdlet, 77
Export-CSV cmdlet, 73–76
extensible providers, 79

F

files
.csv. See .csv (Comma Separated Value) files
text. See text files
File System provider, 86–87
filter boxes, output grids, 66–67
filtering
output, 46–51
before sorting, 50–51
by dates, 47–49
to the left, 49–50
WMI classes, 130–132
Filter parameter, 124, 139
finding cmdlets, Get-Command cmdlet, 30–33
FindLargeDocs.ps1 script, 196
Fix it Center, 191
Force parameter, Update-Help cmdlet, 19–20
forcing intentional errors, 24
ForEach-Object cmdlet, 159, 173
ForEach statement, scripts, 172–175
ForEndlessLoop.ps1 script, 171
For...Loop, creating, 170–172
Format-IPOutput function, 200
Format-List cmdlet, 54, 58–61, 111
Format-NonIPOutput function, 200
Format-Table cmdlet, 53–57
formatting output
lists, 58–61
properties by name, 59
properties by wildcard, 59–61
output grids, 63–67
column buttons, 64–66
filter box, 66–67
tables, 53–57
display, 55–57
ordering properties, 54–55
wide displays, 61–63
AutoSize parameter, 61–62
customizing Format-Wide output, 62–63
Format-Wide cmdlet, 61–63
For statement, scripts, 170–172

ft alias, 134
Full parameter, Get-Help cmdlet, 27
Function provider, 88–89
functions
AddTwo, 211
DivideNum, 211
ease of modification, 196–201
encapsulating business logic, 194–196
Get-AllowedComputer, 224
multiple input parameters, 192–194
type constraints, 190–192
understanding, 183–190
creating functions, 185
naming functions, 185
variable scope, 188
verbs, 187–188
fundamentals
writing scripts, 155–161
enabling support, 156–157
running scripts, 155, 159–160
transitioning from command line to, 157–159
variables/constants, 160–161

G

GC alias, 165
gcim alias, 133
generating random numbers, Get-Random cmdlet, 12
Maximum parameter, 15–16
parameter sets, 17–18
Get-Alias cmdlet, 165
Get-AllowedComputer function, 224
Get-AllowedComputers.ps1 script, 224
Get-AppxPackage cmdlet, 50–51
Get-ChildItem cmdlet, 80, 88, 90
Get-ChoiceFunction.ps1 script, 220
Get-CimAssociatedInstance cmdlet, 134, 138
Get-CimClass cmdlet, 127
Get-CimInstance cmdlet, 132
Get-Command cmdlet, 30–33, 183–185
Get-Content cmdlet, 87
Get-Credential cmdlet, 108
Get-Culture cmdlet, 10–11
Get-Date cmdlet, 11
Get-DirectoryListing function, 193
Get-DirectoryListingToday.ps1 script, 193
Get-Discount function, 194
Get-Doc function

Get-Doc function, 196
Get-EnabledBreakpointsFunction.ps1 script, 214
Get-EventLog cmdlet, 49–50
Get-Help cmdlet, 26–28
Get-Hotfix cmdlet, 8, 13–14
GetInfoByZip method, 191
GetIPDemoSingleFunction.ps1 script, 197
Get-IObjectDefaultEnabledFormatNonIPOutput.ps1 script, 200
Get-IObjectDefaultEnabled.ps1 script, 199
Get-IObject function, 200
Get-ISESnippet cmdlet, 150
Get-ItemProperty cmdlet, 90
Get-Member cmdlet, 33–34
Get-NetAdapter cmdlet, 9, 42
Get-NetConnectionProfile cmdlet, 10
Get-OperatingSystemVersion.ps1 script, 185
Get-Process cmdlet, 7–8, 42, 54
Get-PSDrive cmdlet, 89
Get-PSProvider cmdlet, 80
Get-Random cmdlet, 12, 17–18
Get-Service cmdlet, 9, 43
Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1 script, 189
Get-TextStatisticsCallChildFunction.ps1 script, 188
Get-TextStatistics function, 186
Get-UICulture cmdlet, 11
Get-VM cmdlet, 42
Get-Volume function, 69, 70
Get-WindowsDriver function (DISM), 47
Get-WinEvent cmdlet, 27–28
Get-WmiObject cmdlet, 114
Get-WmiProvider function, 115
grave character, 158
grids, output, 63–67
  column buttons, 64–66
  filter boxes, 66–67
grouping output, 44–46
Group-Object cmdlet, 44–46
gwmi alias, 116

H

handling errors
  limiting choices, 220–225
    contains operator, 223–225
    PromptForChoice method, 220–221
    Test-Connection cmdlet, 222
  missing parameters, 217–220
    creating default value, 218–220
    making parameters mandatory, 219–220
    missing rights, 225–226
    Try/Catch/Finally block, 227–228
Help options, 19–20
Help topics, 26
hotfixes, Get-Hotfix cmdlet, 8
HSG registry key, 92

icm alias, 105
identifying network adapters, Get-NetAdapter cmdlet, 9
If...Else...End construction (VBScript), 177
If statement, scripts, 175–179
  assignment and comparison operators, 177–178
  multiple conditions, 178–179
If...Then...End If statements (VBScript), 175
Import-Clixml cmdlet, 77
Import-Module cmdlet, 83
infrastructure, WMI (Windows Management Instrumentation), 113
InLineGetIPDemo.ps1 script, 196
InputObject parameter (cmdlets), as part of parameter set, 17–18
input parameters, functions, 192–194
Integrated Scripting Environment. See ISE
intellisense, 146–148
intentional errors, forcing, 24
interactive Windows PowerShell console, 142
Invoke-Command cmdlet, 105
Invoke-Item cmdlet, 83
ISE (Integrated Scripting Environment), 2
  navigation, 142–144
  running, 141–148
  Script pane, 145–147
  snippets, 148–151
    creating, 149–150
    creating code with, 148–149
    removing, 150–151
  tab expansion and Intellisense, 146–148
$noun variable

K
keywords, Trap, 191

L
Language Code ID number (LCID), 10
launching
ISE, 141–142
ISE snippets, 148
LCID (Language Code ID number), 10
left, filtering to, 49–50
leveraging providers. See providers
limiting choices, error handling, 220–225
contains operator, 223–225
PromptForChoice method, 220–221
Test-Connection cmdlet, 222
line number, setting breakpoints, 204–205
Line parameter, 204
listing breakpoints, 213–215
ListProcessesSortResults.ps1 script, 154
lists, formatting output, 58–61
properties by name, 59
properties by wildcard, 59–61
literal strings, 163–164
LocalMachine scope (execution policy), 36, 156
logic
business, 194–196
program, 194
LogName parameter, 59
looping technology
Do...Loop statement, 168–169
Do...Until statement, 168
Do...While statement, 165–167

M
Mandatory parameter attribute, 219
MandatoryParameter.ps1 script, 219
mandatory parameters, 219–220
matching Switch statements, scripts, 180–182
Maximum parameter (cmdlets), 15–16
MemberType parameter (cmdlets), 34–35
methods
GetInfoByZip, 191
PromptForChoice, 220–221
Microsoft Fix it Center, 191
Microsoft Management Console (MMC), 83
missing parameters, error handling, 217–220
creating default value, 218–220
making parameters mandatory, 219–220
missing registry property values, 95–96
missing rights, error handling, 225–226
MMC (Microsoft Management Console), 83
Mode parameter, 206
modifying functions, 196–201
multiple conditions, If statements, 178–179
multiple input parameters, functions, 192–194
non-elevated users, security issues, 4–5
non-terminating errors, 226
NotAfter property, 84
Notepad
scripts. See scripts
taxscript log file, 25
NoTypeInformation parameter, 73–75
$noun variable, 223
objects

DateTime, 48
ProcessThreadCollection, 76
WMI (Windows Management Instrumentation), 114

offline analysis technique, 75–76

Online switch, displaying Help information, 19

operators

assignment, 177–178
comparison, 176–178
concatenation, 159
contains, 223–225
range, 17, 166
redirect and append, 70–71
redirect and overwrite, 71–72

ordering properties, tables, 54–55
OutBuffer parameter (cmdlets), 22
Out-File cmdlet, 72
Out-GridView cmdlet, 63–67

output

filtering, 46–51
formatting
lists, 58–61
output grids, 63–67
tables, 53–57
wide displays, 61–63
grouping, 44–46
sorting, 42–44
storing
.csv files, 73–76
text files, 69–73
XML, 76–78

output grids, creating, 63–67
column buttons, 64–66
filter box, 66–67

OutVariable parameter (cmdlets), 22
overwriting information, text files, 71–72

P

pagers, displaying Help information, 27–28

parameters

Action, 209
AutoSize, 56, 61–62
classname, 128
cmdlets, 22–24

EntryPoint, 50
ErrorAction, 23–24
NoElement switched, 46
single parameters, 12–16
Verbose, 22–23
Column, 63
Command, 209–211
computername, 105
ErrorAction, 158
error handling, 217–220
creating default value, 218–220
making parameters mandatory, 219–220

ExpiringInDays, 84
Filter, 124, 139
Force, Update-Help cmdlet, 19–20
intellisense display, 147–148
ISE snippets
Description, 149
Text, 149
Title, 149
Line, 204
LogName, 59
Mode, 206
multiple input parameters and functions, 192–194
NoTypeInformation, 73–75
PassThru, 158
PSComputerName, 105
qualifier, 130–132
Quiet, 222
Recurse, 88
resultclassname, 138
Script, 204
Subject, 85
Value, 93
wildcard asterisk (*), 82
Wrap, 57
$zip input, 191

parameter sets, cmdlets, 16–18

Param statement, assigning default value to
cmdlet parameter, 218–219

Param statements, 193
PassThru parameter, 158
persisted connections, remoting, 107–110
pipeline, 41
filtering output, 46–51
grouping output after sorting, 44–46
sorting output from a cmdlet, 42–44
Pop-Location cmdlet, 92, 94
Q
qualifier parameter, 130–132
queries, WMI (Windows Management Instrumentation), 117–125
return a specific instance, 123–124
return only a few properties, 125–126
return only properties interested in, 122–123
select query, 120–122
Select-String cmdlet, 118–120
Quiet parameter, 222
R
random numbers, generating with Get-Random cmdlet, 12
Maximum parameter, 15–16
parameter sets, 17–18
range operator, 17, 166
Read-Host cmdlet, 186
Recurse parameter, 88
redirect and append operator, 70–71
redirect and overwrite operator, 71–72
redirecting information, text files, 70–72
registry keys
creating, 92–93
setting default value, 93–94
Registry, Registry provider, 89–96
creating registry keys, 92–93
drives, 89–90
missing registry property values, 95–96
modifying registry property value, 94–95
New-Item cmdlet, 94
retrieving registry values, 90–92
setting default value for the key, 93–94
remote procedure call (RPC), 99
RemoteSigned level (scripting support), 156
RemoteSigned level (script support), 36
remoting, 99–112
classic, 99–101
commands, 103–107
configuring, 101–103
persisted connections, 107–110
troubleshooting, 110–111
Remove-Item cmdlet, 81, 150
Remove-PSBreakpoint cmdlet, 215
Remove-PSSession cmdlet, 107
removing snippets (ISE), 150–151
Resolve-ZipCode.ps1 script, 191
resources, WMI (Windows Management Instrumentation), 113
responding to breakpoints, 211–212
Restricted level (script support), 36, 156
resultclassname parameter, 138
retrieving information, Get cmdlets, 6–12
Get-Culture, 10–11
Get-Date, 11
Get-Hotfix, 8
Get-NetAdapter, 9
Get-NetConnectionProfile, 10
Get-Process, 7–8
Get-Random, 12
Get-Service, 9
retrieving registry values, Registry provider, 90–92
RPC (remote procedure call), 99
running
ISE (Integrated Scripting Environment), 141–148
scripts, 155, 159–160

S

Save-Help cmdlet, 20
ScheduledTasks module, 154
scheduled tasks, scripts, 154
script blocks, 49, 162
Script pane (ISE), 145–147
Script parameter, 204
scripts
AccountsWithNoRequiredPassword.ps1, 154
BadScript.ps1, 206
BusinessLogicDemo.ps1, 194
debugging, 203–216
cmdlets, 203–204
deleting breakpoints, 215–216
listing breakpoints, 213–215
responding to breakpoints, 211–212
setting breakpoints, 204–211
DemoBreakFor.ps1, 174
DemoDoWhile.ps1, 166–168
DemoDoWhile.vbs, 165
DemoExitFor.ps1, 175
DemoExitFor.vbs, 174
DemoForEachNext.vbs, 172
DemoForEach.ps1, 173
DemoForLoop.ps1, 170
DemoForLoop.vbs, 170
DemoForWithoutInitOrRepeat.ps1, 170
demolfElseElse.ps1, 170
DemolfElseElse.vbs, 178
DemoIf.ps1, 176
DemoIf.vbs, 176
DemoSwitchArrayBreak.ps1, 182
DemoSwitchArray.ps1, 181
DemoSwitchCase.ps1, 180
DemoSwitchMultiMatch.ps1, 181
DemoTrapSystemException.ps1, 191–192
DemoWhileLessThan.ps1, 162
DirectoryListWithArguments.ps1, 154
DisplayCapitalLetters.ps1, 167
Do...Loop statement, 168–170
DotSourceScripts.ps1, 198
Do...Until statement, 168
DoWhileAlwaysRuns.ps1, 169–170
Do...While statement, 165–168
casting to ASCII values, 167–168
operating over an array, 166–167
range operator, 166
EndlessDoUntil.ps1, 169
FindLargeDocs.ps1, 196
ForEach statement, 172–175
ForEndlessLoop.ps1, 171
For statement, 170–172
fundamentals, 155–161
enabling support, 156–157
running scripts, 155, 159–160
transitioning from commandline to, 157–159
variables/constants, 160–161
Get-AllowedComputers.ps1, 224
Get-ChoiceFunction.ps1, 220
Get-DirectoryListingToday.ps1, 193
Get-EnabledBreakpointsFunction.ps1, 214
GetIPDemoSingleFunction.ps1, 197
Get-IPObjectDefaultEnabledFormatNonIPOutput.ps1, 200
Get-IPObjectDefaultEnabled.ps1, 199
Get-OperatingSystemVersion.ps1, 185
Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1, 189
Get-TextStatisticsCallChildFunction.ps1, 188
If statement, 175–179
assignment and comparison operators, 177–178
multiple conditions, 178–179
Switch syntax, creating an ISE snippet

- Description parameter, 13–14
- Maximum parameter, 15–16
- Name parameter, 14–15
- Site property, 60
- snippets (ISE), 148–151
- creating, 149–150
- creating code with, 148–149
- removing, 150–151
- sorting output, 42–44
- Sort-Object cmdlet, 76
- sorting dates, 47–48
- sorting output from a cmdlet, 42
- special variables, associated meanings, 161
- starting
  - ISE, 141–142
  - ISE snippets, 148
  - transcripts, 24
- Stop-Transcript cmdlet, 25
- stopping transcripts, 25
- .csv files, 73–76
- NoTypeInformation parameter, 73–75
- type information, 75–76
- text files, 69–73
- control, 72–73
- redirecting and appending information, 70–71
- redirecting and overwriting information, 71–72
- XML, 76–78
- strings
  - expanding, 163
  - literal, 163–164
- Subject property, 83
- support, scripts, 156–157
- Switch statement, scripts, 179–182
  - controlling matching behavior, 182
  - Default condition, 180
  - evaluating arrays, 181
  - matching, 180–181
- transcripts, 24
- Start-VM cmdlet, 42
- Start window, accessing PowerShell, 2–4
- Status property, 66
- StopNotepad.ps1 script, 157
- StopNotepadSilentlyContinuePassThru.ps1 script, 159
- StopNotepadSilentlyContinue.ps1 script, 158
- InLineGetIPDemo.ps1, 196
- ListProcessesSortResults.ps1, 154
- MandatoryParameter.ps1, 219
- profiles, 37–38
- reasons for writing, 153–154
- Resolve-ZipCode.ps1, 191
- Set-ExecutionPolicy cmdlet, 36–38
- StopNotepad.ps1, 157
- StopNotepadSilentlyContinuePassThru.ps1, 159
- StopNotepadSilentlyContinue.ps1, 158
- Switch statement, 179–182
  - controlling matching behavior, 182
    - Default condition, 180
    - evaluating arrays, 181
    - matching, 180–181
- Test-ComputerPath, 222
- TestTryCatchFinally.ps1, 227–228
- WhileDoesNotRun.ps1, 170
- WhileReadLine.ps1, 164
- WhileReadLineWend.VBS, 164
- While statement, 162–165
  - construction, 162–164
  - practical example, 164
- Write statement, 164–165
- searching specific certificates, 83–84
- security issues, 4–6
  - administrator rights, 5–6
  - non-elevated users, 4–5
- Select Case statement (VBScript), 179
- Select-Object cmdlet, 115
- select query, WMI (Windows Management Instrumentation), 120–122
- Select-String cmdlet, 118–120
- Set-ExecutionPolicy cmdlet, 36–38, 156–157, 226
- Set-Item cmdlet, 93
- Set-ItemProperty cmdlet, 94
- Set-Location cmdlet, 81, 87, 92
- Set-PropertyItem cmdlet, 94
- Set-PSBreakpoint cmdlet, 204
- Set-PSDebug cmdlet, 203
- setting breakpoints, debugging scripts, 204–211
  - commands, 209–211
  - line number, 204–205
  - variables, 206–209
- Show Command Add-On, 2
- Show-Command cmdlet, 34–35
- SilentlyContinue value (ErrorAction parameter), 158
- single parameters, cmdlets, 12–16
Tab Completion feature

T
Tab Completion feature, 7
tab expansion, 146–148
tables, formatting output, 53–57
display, 55–57
ordering properties, 54–55
terminating errors, 226
Test-ComputerPath script, 222
Test-Connection cmdlet, 222
Test-Path cmdlet, 38, 92, 95
Test registry key, 93
TestTryCatchFinally.ps1 script, 227–228
Test-WSMan cmdlet, 103, 110
text files, storing output, 69–73
control, 72–73
redirecting and appending information, 70–71
redirecting and overwriting information, 71–72
Text parameter, snippets, 149
time, retrieving with Get-Date cmdlet, 11
Title parameter, snippets, 149
transcript log files, Notepad, 25
transcripts
starting, 24
stopping, 25
Trap keyword, 191
Try/Catch/Finally blocks, 227–228
type constraints, functions, 190–192
type information, .csv files, 75–76

U
UAC (User Account Control) dialog box, 5
UAC (User Account Control) feature, 225
UI (user interface) culture settings, 11
Undefined level (script support), 36, 156
Unrestricted level (script support), 36, 156
Update-Help cmdlet, Force parameter, 19–20
updating Help, 19–20
User Account Control (UAC) dialog box, 5
User Account Control (UAC) feature, 225
user-defined snippets (ISE), 149–151
user interface (UI) culture settings, 11
users
administrators, security issues, 5–6
non-elevated, security issues, 4–5

V
Value parameter, 93
values, ErrorAction parameter, 24
Variable provider, 96–97
variables
$_ary, 173
$_ automatic, 159
$_caps, 167
$_caption, 220
$_choiceRTN, 221
$_cn, 105
$_cred, 105
$_dc1, 107
$_discount, 195
$_fileContents, 164
$_logon, 135
$_message, 220
$_noun, 223
$_obj1, 227
$_process, 159
scripts, 160–161
setting breakpoints, 206–209
special variables with associated meanings, 161
variable scope, 188
VBScript
Exit For statements, 174
For...Each...Next construction, 172
For...Next...Loop, 170
If...Else...End construction, 177
If...Then...End If statements, 175
Select Case statement, 179
While...Wend loop, 162
Wscript.Quit statements, 175
Verbose parameter (cmdlets), 22–23
verbs, functions, 187–188

W
WarningAction parameter (cmdlets), 22
WarningVariable parameter (cmdlets), 22
WhatIf parameter (cmdlets), 22
Where-Object filter, 46–47
WhileDoesNotRun.ps1 script, 170
WhileReadLine.ps1 script, 164
WhileReadLineWend.VBS script, 162
While statement, scripts, 162–165
construction, 162–164
practical example, 164
While...Wend loop (VBScript), 162
wide display
formatting output
AutoSize parameter, 61–62
customizing Format-Wide output, 62–63
lists, 61
wild card asterisk (*) parameter, 82
wildcard search pattern, Command Add-On, 143
wildcards, selecting properties to display, 59–61
Win32_PingStatus WMI class, 222
Win32_Process cmdlet, 135
windows
Command Add-on, 142–143
Start, accessing PowerShell, 2–4
Windows Management Instrumentation. See WMI
Windows Remote Management (WinRm), 101
WinRm (Windows Remote Management), 101
WMI (Windows Management Instrumentation),
113–125
classes, 115–117
classes, exploring with CIM cmdlets, 127–132
consumers, 113
filtering classes with CIM cmdlets, 130–132
finding class methods, CIM, 128–129
infrastructure, 113
objects and namespaces, 114
providers, 114–115
queries, 117–125
return a specific instance, 123–124
return only a few properties, 125–126
return only properties interested in, 122–123
select query, 120–122
Select-String cmdlet, 118–120
resources, 113
retrieving instances, CIM, 132
Wrap parameter, 57
Write-Host cmdlet, 209
Write-Path function, 188
Write statement, scripts, 164–165
writing scripts. See scripts
Wscript.Quit statements (VBScript), 175

X
XML, storing output, 76–78

Z
$zip input parameter, 191
Now that you’ve read the book...

Tell us what you think!

Was it useful?
Did it teach you what you wanted to learn?
Was there room for improvement?

Let us know at http://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press, and we read every one of your responses. Thanks in advance!