

Windows PowerShell 3.0

First Steps

Windows PowerShell 3.0 First Steps

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!"

ISBN: 978-0-7356-8100-2



9 0 0 0 0

U.S.A. \$29.99

Canada \$31.99

[Recommended]

Operating Systems/Windows Server



Windows PowerShell 3.0 First Steps

Ed Wilson

Copyright © 2013 by Ed Wilson

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-8100-2

1 2 3 4 5 6 7 8 9 LSI 8 7 6 5 4 3

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at <http://www.microsoft.com/learning/booksurvey>.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Developmental Editor: Michael Bolinger

Production Editor: Melanie Yarbrough

Editorial Production: Box Twelve Communications

Technical Reviewer: Brian Wilhite

Indexer: Box Twelve Communications

Cover Design: Twist Creative • Seattle

Cover Composition: Ellie Volckhausen

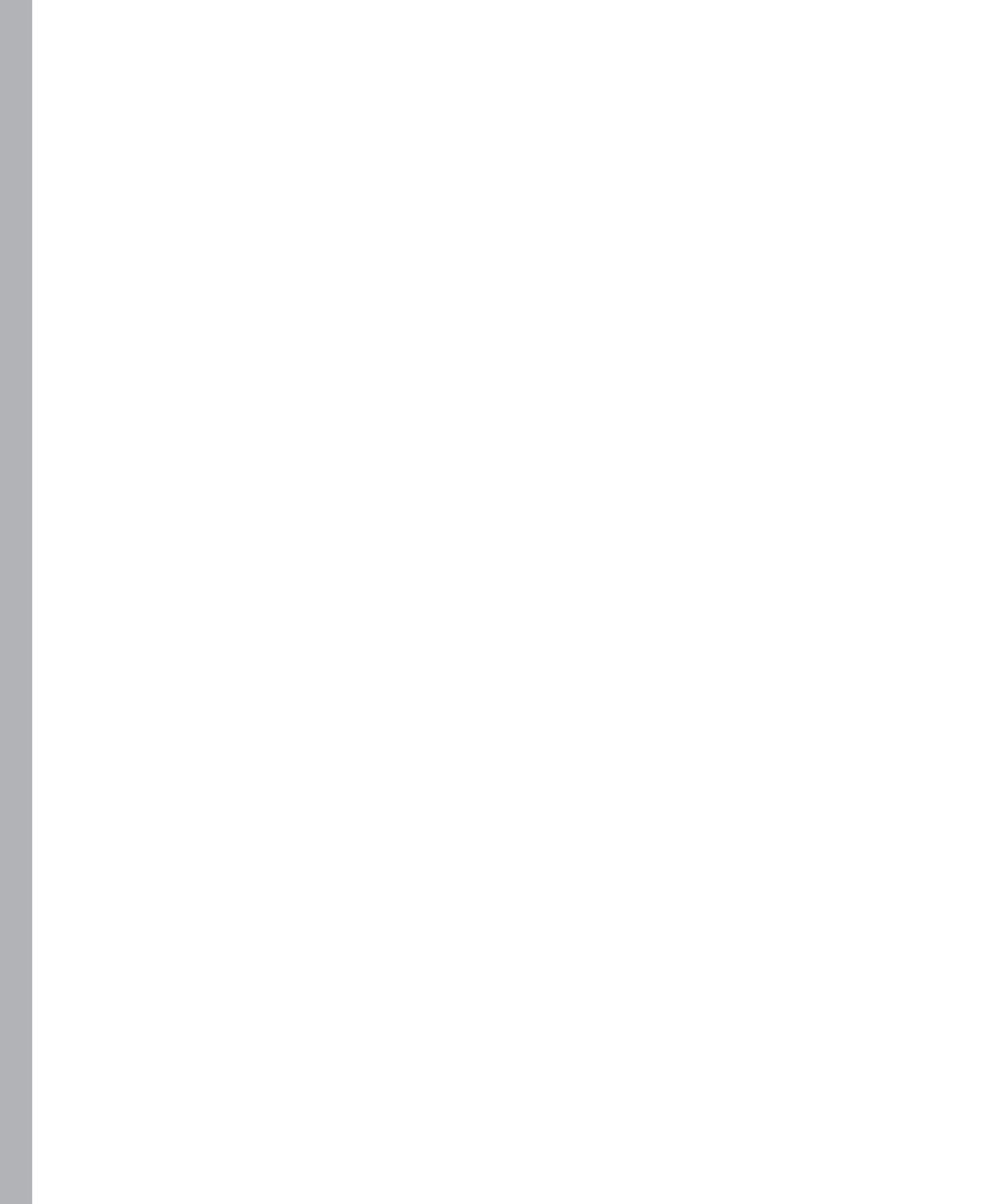
Illustrator: Rebecca Demarest

To Teresa, my soul mate.

—ED WILSON

Contents at a glance

	<i>Foreword</i>	<i>xv</i>
	<i>Introduction</i>	<i>xvii</i>
CHAPTER 1	Overview of Windows PowerShell 3.0	1
CHAPTER 2	Using Windows PowerShell cmdlets	21
CHAPTER 3	Filtering, grouping, and sorting	41
CHAPTER 4	Formatting output	53
CHAPTER 5	Storing output	69
CHAPTER 6	Leveraging Windows PowerShell providers	79
CHAPTER 7	Using Windows PowerShell remoting	99
CHAPTER 8	Using WMI	113
CHAPTER 9	Using CIM	127
CHAPTER 10	Using the Windows PowerShell ISE	141
CHAPTER 11	Using Windows PowerShell scripts	153
CHAPTER 12	Working with functions	183
CHAPTER 13	Debugging scripts	203
CHAPTER 14	Handling errors	217
APPENDIX A	Windows PowerShell FAQ	229
APPENDIX B	Windows PowerShell 3.0 coding conventions	239
	<i>Index</i>	<i>247</i>



Contents

<i>Foreword</i>	<i>xv</i>
<i>Introduction</i>	<i>xvii</i>

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: <i>Get</i>	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 <i>Get-Help</i> 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:

www.microsoft.com/learning/booksurvey/

Using the <i>Get-Command</i> to find cmdlets.	30
Using the <i>Get-Member</i> cmdlet.	33
Exploring property members	34
Using the <i>Show-Command</i> 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 <i>-AutoSize</i> parameter to configure the output	61
Customizing the <i>Format-Wide</i> output	62

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

Configuring Windows PowerShell remoting	101
Running commands	103
Creating a persisted connection	107
Troubleshooting Windows PowerShell remoting	110
Summary	111
Chapter 8 Using WMI	113
Understanding the WMI Model	113
Working with objects and namespaces	114
Listing WMI providers	114
Working with WMI classes	115
Querying WMI: The basics	117
Tell me everything about everything	120
Tell me selected things about everything	122
Tell me everything about some things	123
Tell me selected things about some things	125
Summary	125
Chapter 9 Using CIM	127
Using CIM cmdlets to explore WMI classes	127
Using the classname parameter	128
Finding WMI class methods	128
Filtering classes by qualifier	130
Reducing returned properties and instances	133
Cleaning up output from the command	134
Working with associations	134
Summary	140
Chapter 10 Using the Windows PowerShell ISE	141
Running the Windows PowerShell ISE	141
Navigating the Windows PowerShell ISE	142
Working with the Script pane	145
Tab expansion and Intellisense	146

Working with Windows PowerShell ISE snippets	148
Using Windows PowerShell ISE snippets to create code	148
Creating new Windows PowerShell ISE snippets	149
Removing user-defined Windows PowerShell ISE snippets	150
Summary.	151

Chapter 11 Using Windows PowerShell scripts 153

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 <i>While</i> statement	162
Constructing the <i>While</i> statement	162
A practical example of using the <i>While</i> statement	164
Using special features of Windows PowerShell	164
Using the <i>Do...While</i> statement	165
Using the <i>range</i> operator	166
Operating over an array	166
Casting to ASCII values	167
Using the <i>Do...Until</i> statement	168
Using the Windows PowerShell <i>Do...Loop</i> statement	168
Using the <i>For</i> statement	170
Creating a <i>For...Loop</i>	170
Using the <i>ForEach</i> statement	172
Exiting the <i>ForEach</i> statement early	174
Using the <i>If</i> statement	175
Using assignment and comparison operators	177
Evaluating multiple conditions	178

Using the <i>Switch</i> statement.	179
Using the basic <i>Switch</i> 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

Limiting choices.	220
Using PromptForChoice to limit selections	220
Using <i>Test-Connection</i> to identify accessible computers	222
Using the <i>contains</i> operator to examine contents of an array	223
Handling missing rights.	225
Attempting and failing	226
Checking for rights and exiting gracefully	226
Using <i>Try/Catch/Finally</i>	227
Summary.	228

Appendix A Windows PowerShell FAQ 229

Appendix B Windows PowerShell 3.0 coding conventions 239

General script construction.	239
Include functions in the script that uses the functions	239
Use full cmdlet names and full parameter names	240
Use <i>Get-Item</i> to convert path strings to rich types	241
General script readability.	241
Formatting your code	242
Working with functions	244
Creating template files	244
Writing your own functions	245
Variables, constants, and naming	245

<i>Index</i>	247
--------------	-----

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:

www.microsoft.com/learning/booksurvey/

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.
- Windows PowerShell 3.0 is already installed on Windows 8 and on Windows Server 2012. You can obtain evaluation versions of those operating systems from TechNet:
<http://technet.microsoft.com/en-US/evalcenter/hh699156.aspx?ocid=wc-tn-wctc>
http://technet.microsoft.com/en-US/evalcenter/hh670538.aspx?wt.mc_id=TEC_108_1_4
- The section on Active Directory requires access to Active Directory Domain Services. For those examples, ensure you have access to Windows Server 2012.
- For the chapter on Exchange server, you need access to a server running Microsoft Exchange Server 2013. You can obtain an evaluation version of that from TechNet:
<http://technet.microsoft.com/en-us/evalcenter/hh973395.aspx>

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.

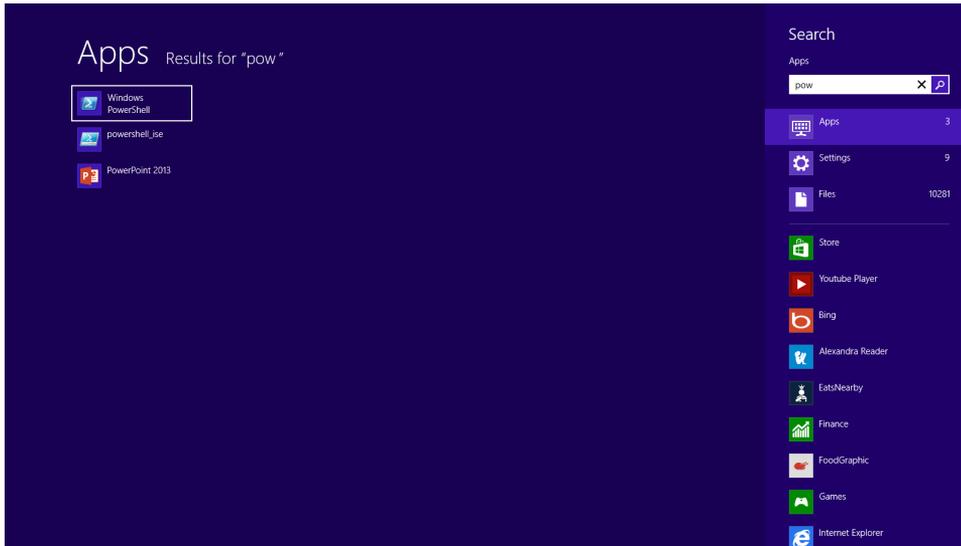


FIGURE 1-1 Typing in the Start window opens the Search window highlighting the Windows PowerShell console.

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.

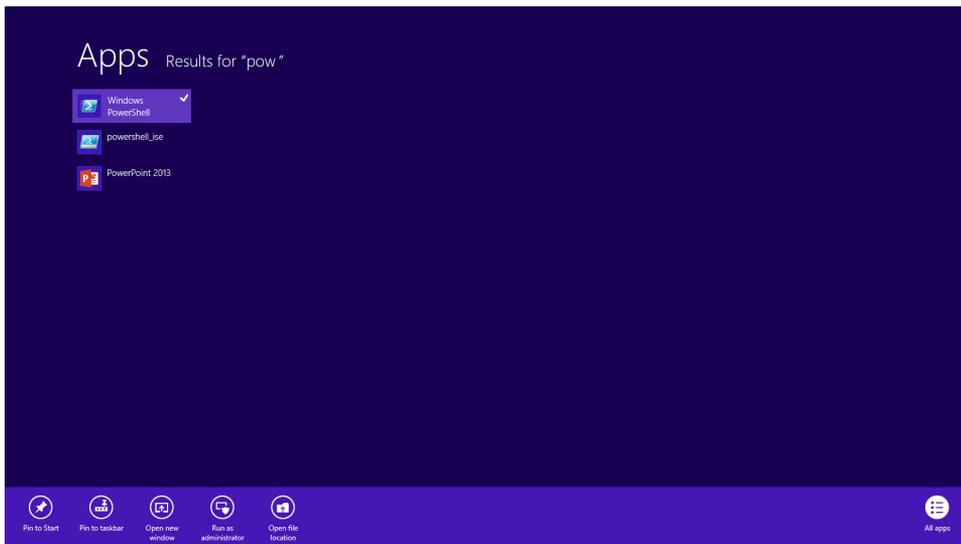


FIGURE 1-2 By right-clicking the Windows PowerShell icon in the Search results box, the Pin to Start and the Pin to taskbar options appear.

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/S
torage/MSFT_Disk) [Get-Disk], CimException
+ FullyQualifiedErrorId : MI_RESULT 2,Get-Disk
```

TIP 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.

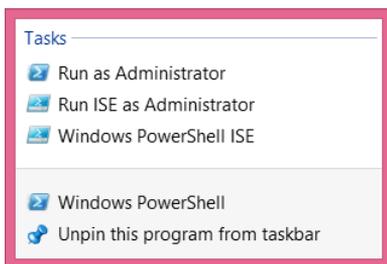


FIGURE 1-3 Right-click the Windows PowerShell icon to bring up the option to Run as Administrator.

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.

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
99	10	1676	4868	62		2232	BtwRSupportService
84	9	1532	4312	46		2504	CamMute
35	6	912	2908	48	0.00	1180	conhost
33	5	748	2364	26		1832	conhost
52	8	1788	5832	54	0.27	4944	conhost
375	14	1764	3288	48		628	csrss
364	23	2064	46212	90		732	csrss
114	9	1424	4476	54		2300	CxAudMsg64
179	15	4704	7400	69		4240	daemonu
335	33	33712	44428	277		1096	dwm
293	22	5756	12804	101		2340	EvtEng
1847	125	76296	131640	861	59.98	4216	explorer
31	8	1160	3504	48	0.83	4156	fmapp
96	9	1552	5164	78	0.03	4188	hkcmd
323	29	35708	40748	250		3024	IAStorDataMgrSvc
268	23	21820	25208	244	0.14	5636	IAStorIcon
70	8	1072	3160	30		980	ibmpmsvc
0	0	0	20	0		0	Idle
125	12	2036	6516	86	0.02	4180	igfxpers
461	39	13564	12244	791	0.47	4584	LiveComm
272	33	29256	32636	568		3032	LnvHotSpotSvc
306	27	17700	23968	170		4804	loctaskmgr
210	17	9828	14112	153	0.08	4912	lpdagent
881	26	4336	9500	38		824	lsass
131	12	1980	6696	79	0.00	3388	MobileHotspotClient
471	85	77616	54784	248		2876	MsMpEng
106	10	2652	5904	39		456	nvSCPAPISvr

FIGURE 1-4 The Windows PowerShell *Get-Process* cmdlet returns detailed Windows process information.

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
```

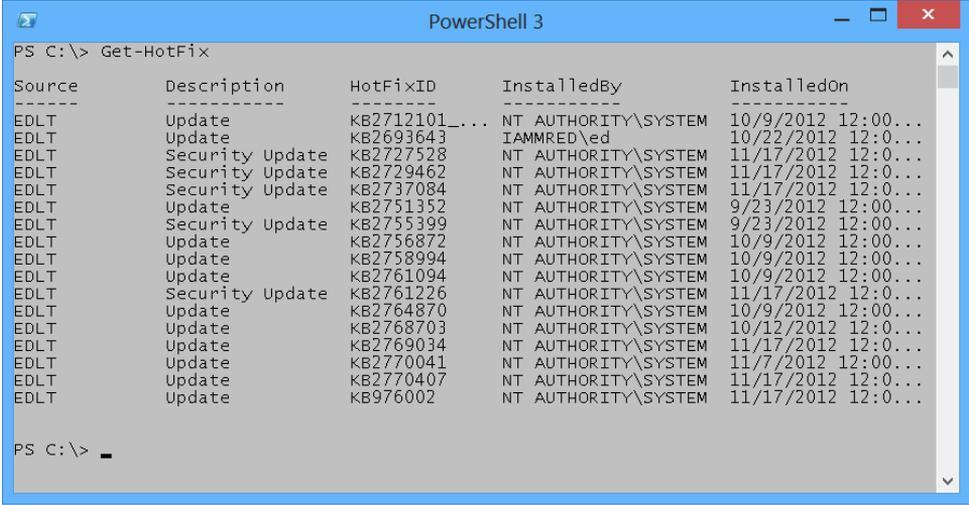
NOTE It is a Windows PowerShell convention to use singular nouns. While not universally applied (my computer has about 50 plural nouns), it is a good place to start. So if you are not sure if a noun (or parameter) is singular or plural, choose the singular. Most of the time you will be correct.

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.



```
PS C:\> Get-HotFix

Source      Description      HotFixID      InstalledBy      InstalledOn
-----
EDLT       Update           KB2712101...  NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT       Update           KB2693643    IAMMRED\ed      10/22/2012 12:0...
EDLT       Security Update  KB2727528    NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT       Security Update  KB2729462    NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT       Security Update  KB2737084    NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT       Update           KB2751352    NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT       Security Update  KB2755399    NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT       Update           KB2756872    NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT       Update           KB2758994    NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT       Update           KB2761094    NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT       Security Update  KB2761226    NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT       Update           KB2764870    NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT       Update           KB2768703    NT AUTHORITY\SYSTEM  10/12/2012 12:0...
EDLT       Update           KB2769034    NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT       Update           KB2770041    NT AUTHORITY\SYSTEM  11/7/2012 12:00...
EDLT       Update           KB2770407    NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT       Update           KB976002     NT AUTHORITY\SYSTEM  11/17/2012 12:0...

PS C:\> _
```

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-Servi
```

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
```

Status	Name	DisplayName
Running	AdobeActiveFile...	Adobe Active File Monitor V6
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Stopped	AllUserInstallA...	Windows All-User Install Agent

<TRUNCATED OUTPUT>

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-NetA
```

The following example shows the command and associated output:

```
PS C:\> Get-NetAdapter
```

Name	InterfaceDescription	ifIndex	Status
Network Bridge	Microsoft Network Adapter Multiplexo...	29	Up
Ethernet	Intel(R) 82579LM Gigabit Network Con...	13	Not Pre...
vEthernet (WirelessSwi...	Hyper-V Virtual Ethernet Adapter #4	31	Up
vEthernet (External Sw...	Hyper-V Virtual Ethernet Adapter #3	23	Not Pre...
vEthernet (InternalSwi...	Hyper-V Virtual Ethernet Adapter #2	19	Up
Bluetooth Network Conn...	Bluetooth Device (Personal Area Netw...	15	Disconn...
Wi-Fi	Intel(R) Centrino(R) Ultimate-N 6300...	12	Up

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:

```
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-UI-Culture* cmdlet. Using tab expansion to call the *Get-UICulture* cmdlet, type the following, then press the Tab key followed by Enter:

```
Get-Ui
```

The following example shows the command and associated output:

```
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:

```
Get-Dat
```

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

```
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.

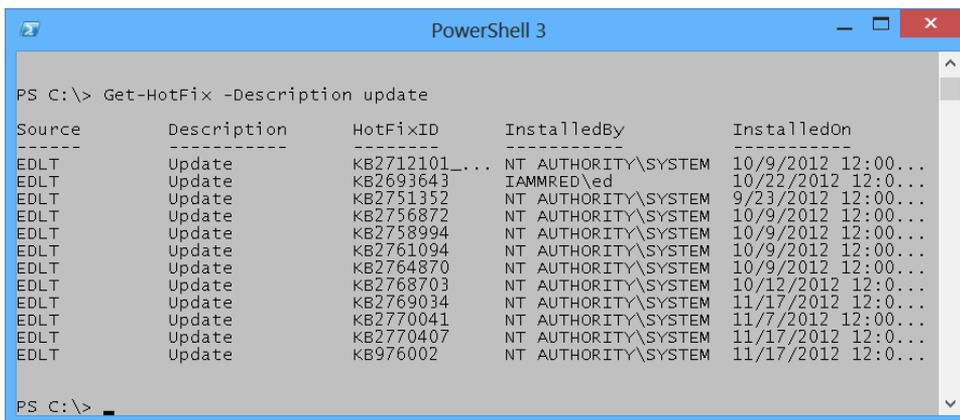
Using single parameters

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.



```
PS C:\> Get-HotFix -Description update

Source      Description      HotFixID      InstalledBy      InstalledOn
-----
EDLT        Update           KB2712101...  NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT        Update           KB2693643     IAMMRED\ed       10/22/2012 12:00...
EDLT        Update           KB2751352     NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT        Update           KB2756872     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT        Update           KB2758994     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT        Update           KB2761094     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT        Update           KB2764870     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT        Update           KB2768703     NT AUTHORITY\SYSTEM  10/12/2012 12:00...
EDLT        Update           KB2769034     NT AUTHORITY\SYSTEM  11/17/2012 12:00...
EDLT        Update           KB2770041     NT AUTHORITY\SYSTEM  11/7/2012 12:00...
EDLT        Update           KB2770407     NT AUTHORITY\SYSTEM  11/17/2012 12:00...
EDLT        Update           KB976002      NT AUTHORITY\SYSTEM  11/17/2012 12:00...
```

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
+ FullyQualifiedErrorId : GetHotFixNoEntriesFound,Microsoft.PowerShell.Commands
.GetHotFixCommand
```

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 positional parameter for the *Stop-Process* 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
+ FullyQualifiedErrorId : PositionalParameterNotFound,Microsoft.PowerShell.Commands.GetRandomCommand
```

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:

```
Get-Random [[-Maximum] <Object>] [-Minimum <Object>] [-SetSeed <Int32>]
[<CommonParameters>]
```

```
Get-Random [-InputObject] <Object[]> [-Count <Int32>] [-SetSeed <Int32>]
[<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.

    Required?                true
    Position?                1
    Default value
    Accept pipeline input?   true (ByValue)
    Accept wildcard characters? False
```

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:

```
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:

```
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
```

<OUTPUT TRUNCATED>

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.

Using Windows PowerShell remoting

- Using PowerShell remoting
- Configuring Windows PowerShell remoting
- Troubleshooting Windows PowerShell remoting

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
```

Name	Synopsis
-----	-----
Add-Computer	Add the local computer to a domain or workgroup.
Add-Printer	Adds a printer to the specified computer.
Add-PrinterDriver	Installs a printer driver on the specified computer.
Add-PrinterPort	Installs a printer port on the specified computer.

<...Output Truncated ...>

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

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

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

TimeCreated	ProviderName	Id	Message
-----	-----	--	-----
7/1/2012 11:54:14 AM	MSEXchange ADAcess	2080	Process MAD.EXE (...)

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 char:1
+ Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential iam
...
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Get-WinEvent], EventLogException
+ FullyQualifiedErrorId : System.Diagnostics.Eventing.Reader.EventLogException, Microsoft.PowerShell.Commands.GetWinEventCommand
```

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

Status Name                DisplayName
----- ----                -
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*.
6. Sets a target listener named *Microsoft.powershell.workflow*.
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?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(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 mac
hine.
WinRM firewall exception enabled.

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name:
microsoft.powershell SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will
allow selected users to remotely run Windows PowerShell commands on this computer".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name:
microsoft.powershell.workflow SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will
allow selected users to remotely run Windows PowerShell commands on this computer".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name:
microsoft.powershell132 SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will
allow selected users to remotely run Windows PowerShell commands on this computer".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(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
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.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\PowerShe
ll_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\PowerShe
ll_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.

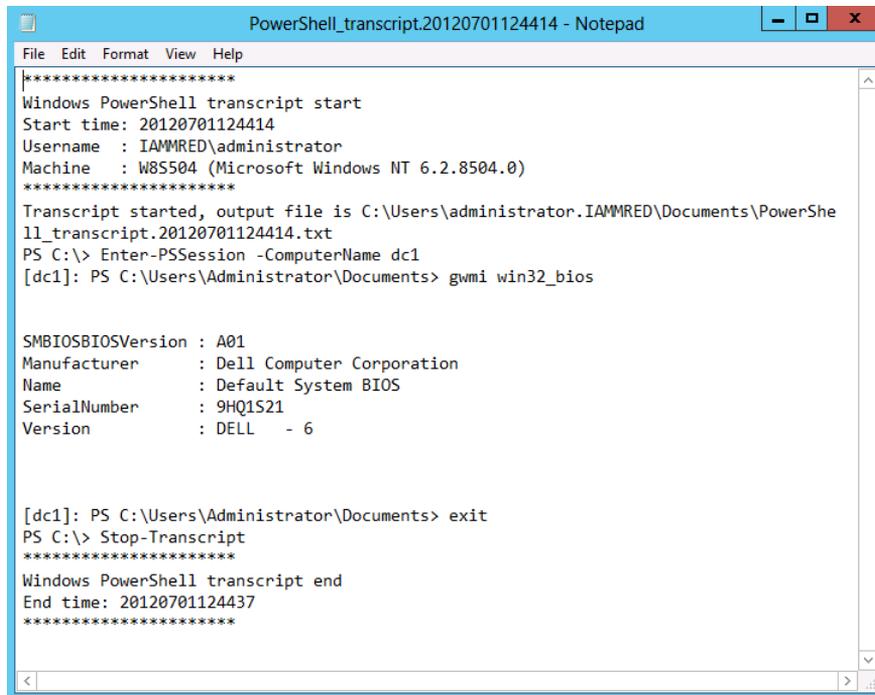


FIGURE 7-1 The transcript tool works in remote Windows PowerShell sessions as well as in local Windows PowerShell console sessions.

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}
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName	PSComputerName
-----	-----	-----	-----	-----	-----	--	-----	-----
224	34	47164	51080	532	0.58	10164	wsmprovhost	ex1

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}
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName	PSComputerName
-----	-----	-----	-----	-----	-----	--	-----	-----
221	34	47260	51048	536	0.33	4860	wsmprovhost	ex1

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","sq11","wsus1","wds1","hyperv1","hyperv2","hyperv3"  
PS C:\> $cred = Get-Credential iammred\administrator  
PS C:\> Invoke-Command -cn $cn -cred $cred -ScriptBlock {gwmi win32_bios}
```

```

SMBIOSBIOSVersion : BAP6710H.86A.0072.2011.0927.1425
Manufacturer       : Intel Corp.
Name              : BIOS Date: 09/27/11 14:25:42 Ver: 04.06.04
SerialNumber      :
Version           : INTEL - 1072009
PSComputerName    : hyperv3

SMBIOSBIOSVersion : A11
Manufacturer       : Dell Inc.
Name              : Phoenix ROM BIOS PLUS Version 1.10 A11
SerialNumber      : BDY91L1
Version           : DELL - 15
PSComputerName    : hyperv2

SMBIOSBIOSVersion : A01
Manufacturer       : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
PSComputerName    : dc1

SMBIOSBIOSVersion : 090004
Manufacturer       : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04
SerialNumber      : 3692-0963-1044-7503-9631-2546-83
Version           : VRTUAL - 3000919
PSComputerName    : wsus1

SMBIOSBIOSVersion : V1.6
Manufacturer       : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : To Be Filled By O.E.M.
Version           : 7583MS - 20091228
PSComputerName    : hyperv1

SMBIOSBIOSVersion : 080015
Manufacturer       : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : None
Version           : 091709 - 20090917
PSComputerName    : sql1

SMBIOSBIOSVersion : 080015
Manufacturer       : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : None
Version           : 091709 - 20090917
PSComputerName    : wds1

SMBIOSBIOSVersion : 090004
Manufacturer       : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04
SerialNumber      : 8994-9999-0865-2542-2186-8044-69
Version           : VRTUAL - 3000919
PSComputerName    : dc3

```

```

SMBIOSBIOSVersion : 090004
Manufacturer       : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04
SerialNumber      : 2301-9053-4386-9162-8072-5664-16
Version           : VIRTUAL - 3000919
PSComputerName    : ex1

```

```
PS C:\>
```

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:

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

Index	Time	EntryType	Source	InstanceID	Message	PSComputerName
17702	Jul 01 12:59	Information	ESENT	701	DFSR...	dc1

```
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.



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:

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

Status	Name	DisplayName	PSComputerName
Stopped	AeLookupSvc	Application Experience	ex1
Running	ADWS	Active Directory Web Services	dc3

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:

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

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName	PSComputerName
47	7	1812	6980	53	0.70	3300	conhost	dc3
32	4	824	2520	22	0.22	1140	conhost	ex1

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

```
Windows PowerShell is Cool!
PS C:\> $cred = Get-Credential -Credential iammred\administrator
PS C:\> $cn = "ex1","dc3"
PS C:\> $ps = New-PSSession -ComputerName $cn -Credential $cred
PS C:\> Invoke-Command -Session $ps -ScriptBlock {gsv | select -First 1}

Status      Name                DisplayName                PSComputerName
-----
Stopped     AeLookupSvc        Application Experience     ex1
Running     ADWS                Active Directory Web Services dc3

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

Handles  NPM(K)  PM(K)  WS(K) VM(M)  CPU(s)  Id ProcessName  PSComputerName
-----
47        7    1812   6980  53    0.70    3300 conhost      dc3
32        4     824   2520  22    0.22    1140 conhost      ex1

PS C:\> _
```

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
```

```
wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.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
```

```
wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.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
```

```
wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.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
```

Id	Name	ComputerName	State	ConfigurationName	Availability
3	Session3	ex1	Opened	Microsoft.PowerShell	Available
4	Session4	dc3	Opened	Microsoft.PowerShell	Available

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):

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

```
State                : Opened
IdleTimeout          : 7200000
OutputBufferingMode  : None
ComputerName         : dc3
ConfigurationName    : Microsoft.PowerShell
InstanceId           : c15cc80d-64f0-4096-a010-0211f0188aec
Id                   : 4
Name                 : Session4
Availability          : Available
ApplicationPrivateData : {PSVersionTable}
Runspace             : System.Management.Automation.RemoteRunspace
```

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:

```
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:

```
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
- \$ary variable, 173
- 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
- \$_ 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

- \$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

classes

- cmdlets, 127–132
 - classname parameter, 128
 - filtering WMI classes by qualifiers, 130–132
 - finding WMI class methods, 128–129
 - reducing returned properties and instances, 133
 - retrieving WMI instances, 132
- classes
 - Win32_PingStatus WMI, 222
 - WMI (Windows Management Instrumentation), 115–117, 127–132
- classic remoting, 99–101
- class methods (WMI), 128–129
- classname parameter, 128
- cmdlets, 6–16
 - About conceptual Help topics, 29–30
 - Add-Content, 87
 - CIM (Common Information Model), 127–132
 - classname parameter, 128
 - filtering WMI classes by qualifiers, 130–132
 - finding WMI class methods, 128–129
 - debugging scripts, 203–204
 - Disable-PSBreakpoint, 215
 - Enable-Netadapter, 42
 - Enable-PSBreakpoint, 215
 - Enter-PSSession, 103, 107
 - Export-Clixml, 77
 - Export-CSV, 73–76
 - ForEach-Object, 159, 173
 - Format-List, 54, 58–61, 111
 - Format-Table, 53–57
 - controlling table display, 55–57
 - Format-Wide, 61–63
 - Get-Alias, 165
 - Get-AppxPackage, 50–51
 - Get-ChildItem, 80, 88, 90
 - Get-CimAssociatedInstance, 134, 138
 - Get-CimClass, 127
 - Get-CimInstance, 132
 - Get-Command, 30–33, 183–185
 - Get-Content, 87
 - Get-Credential, 108
 - Get-Culture, 10–11
 - Get-Date, 11
 - Get-EventLog, 49–50
 - Get-Help, 26–28
 - Get-Hotfix, 8
 - Get-ISESnippet, 150
 - Get-ItemProperty, 90
 - Get-Member, 33–34
 - Get-NetAdapter, 9, 42
 - Get-NetConnectionProfile, 10
 - Get-Process, 7–8, 42
 - Get-PSDrive, 89
 - Get-PSProvider, 80
 - Get-Random, 12
 - Get-Service, 9, 43
 - Get-UICulture, 11
 - Get-VM, 42
 - Get-WinEvent, 27–28
 - Get-WmiObject, 114
 - Group-Object, 44–46
 - Help topics, 26
 - Import-Clixml, 77
 - Import-Module, 83
 - intellisense display, 147–148
 - Invoke-Command, 105
 - Invoke-Item, 83
 - New-Alias, 82
 - New-ISESnippet, 149
 - New-Item, 38, 94
 - New-PSDrive, 89
 - New-PSSession, 107
 - New-WebServiceProxy, 191
 - Out-File, 72
 - Out-GridView, 63–67
 - parameters, 22–24
 - EntryType, 50
 - ErrorAction parameter, 23–24
 - Verbose, 22–23
 - parameter sets, 16–18
 - Pop-Location, 92, 94
 - property members, 34–35
 - Push-Location, 92
 - Read-Host, 186
 - Remove-Item, 81, 150
 - Remove-PSBreakpoint, 215
 - Remove-PSSession, 107
 - Save-Help, 20
 - Select-Object, 115
 - Select-String, 118–120
 - Set-ExecutionPolicy, 36–38, 156–157, 226
 - Set-Item, 93
 - Set-ItemProperty, 94
 - Set-Location, 81, 87, 92
 - Set-PropertyItem, 94
 - Set-PSBreakpoint, 204

- 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
- comparison 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
- control, text files, 72–73
- 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
 - NoTypeInfo 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
- demofIfElseIfElse.ps1 script, 178
- DemofIfElseIfElse.vbs script, 178
- Demof.ps1 script, 176
- Demof.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, 168–170
- 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

E

- 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

I

- 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

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

N

Name parameter (cmdlets), 14–15

Name parameter, Get-Help cmdlet, 26

namespaces, WMI (Windows Management Instrumentation), 114

naming functions, 185

navigation, ISE (Integrated Scripting Environment), 142–144

network adapters, identifying with Get-NetAdapter cmdlet, 9

New-Alias cmdlet, 82

New-IseSnippet cmdlet, 149

New-Item cmdlet, 38, 94

New-PSDrive cmdlet, 89

New-PSSession cmdlet, 107

New-WebServiceProxy cmdlet, 191

NoElement switched parameter, 46

NoExit argument, 159–160

non-elevated users, security issues, 4–5

non-terminating errors, 226

NotAfter property, 84

Notepad

scripts. *See* scripts

transcript log file, 25

NoTypeInfoInformation parameter, 73–75

\$noun variable, 223

O

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

- paggers, displaying Help information, 27–28
- parameters
 - Action, 209
 - AutoSize, 56, 61–62
 - classname, 128
 - cmdlets, 22–24

- EntryType, 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
- NoTypeInfoInformation, 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
 - command-line 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

- positional arguments, 187
- position message, 158
- Process.csv text file, 74
- ProcessInfo.csv file, 76
- Process scope (execution policy), 36, 156
- ProcessThreadCollection object, 76
- \$process variable, 159
- \$profile automatic variable, 37
- profiles, 37–38
- program logic, 194
- PromptForChoice method, 220–221
- properties
 - DateTime object, 48
 - Get-Process cmdlet, 54
 - lists, 59–61
 - NotAfter, 84
 - PSISContainer, 84
 - Site, 60
 - Status, 66
 - Subject, 83
 - tables, 54–55
- property members, 34–35
- provider class, WMI, 115
- providers, 79–97
 - Alias, 80–82
 - Certificate, 82–84
 - Environment, 85–86
 - File System, 86–87
 - Function, 88–89
 - Get-PSProvider cmdlet, 80
 - Registry, 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
 - Variable, 96–97
 - WMI (Windows Management Instrumentation), 114–115
- PSComputerName parameter, 105
- PSISContainer property, 84
- PSScheduledJob module, 154
- PSStatus property set, 120
- Push-Location cmdlet, 92

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)

- 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
 - demoIfElseIfElse.ps1, 170
 - DemoIfElseIfElse.vbs, 178
 - DemoIf.ps1, 176
 - DemoIf.vbs, 176
 - DemoSelectCase.vbs, 179
 - 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

- 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
 - 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
- Start-Transcript cmdlet, 24, 38
- 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
- stopping transcripts, 25
- Stop-Transcript cmdlet, 25
- storing output
 - .csv files, 73–76
 - NoTypeInfoInformation 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 parameter, 85
- 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
- Switch syntax, creating an ISE snippet, 149

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

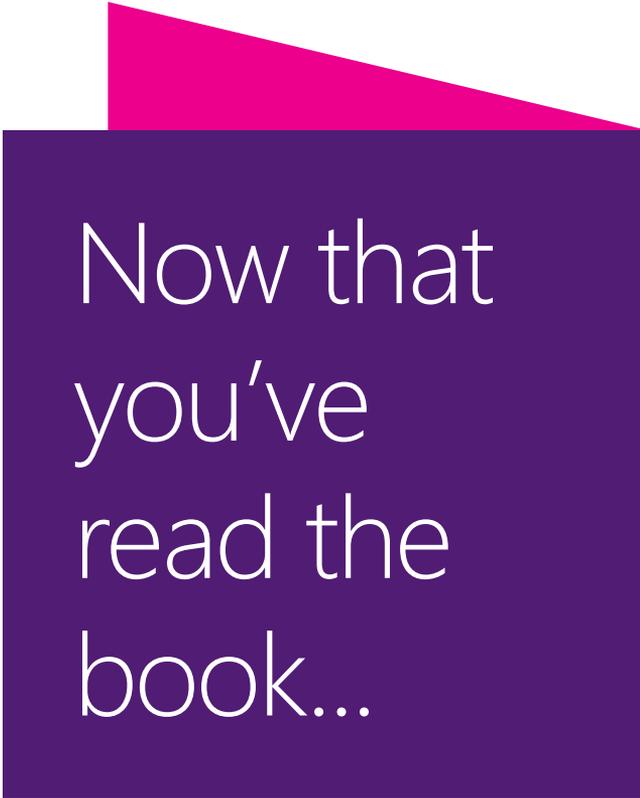
- While statement, scripts, 162–165
 - construction, 162–164
 - practical example, 164
- While...Wend loop (VBScript), 162
- wide display
 - formatting output
 - AutoSize parameter, 61–62
 - customizing Format-Wide output, 62–63
 - lists, 61
- wild card asterisk (*) parameter, 82
- wildcard search pattern, 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!

