Microsoft

Windows® PowerShell 3.0

Ed Wilson

online book + practice files

Step by Step

Windows PowerShell 3.0 Step by Step

Your hands-on, step-by-step guide to automating Windows® administration with Windows PowerShell 3.0

Teach yourself the fundamentals of Windows PowerShell 3.0 command line interface and scripting language—one step at a time. Ideal for those with fundamental programming skills, this tutorial provides practical, learn-by-doing exercises to help you automate maintenance and administrative tasks.

Discover how to:

- Manage local and remote systems using built-in cmdlets
- Write scripts to handle recurring operations
- Concurrently accomplish multiple tasks
- Connect to a remote system and run commands
- Reuse code and simplify script creation
- Manage users, groups, and computers with Active Directory®
- Track down and fix script errors with the Windows PowerShell debugger
- Execute scripts to administer and troubleshoot Microsoft Exchange Server 2010

Your *Step by Step* digital content includes:

- Downloadable practice files
  See http://go.microsoft.com/FWLink/?Linkid=275531
- Fully searchable ebook. *See the instruction page at the back of the book*

ISBN: 978-0-7356-6339-8

microsoft.com/mspress

U.S.A. $54.99
Canada $57.99
[*Recommended*]

Programming/Windows PowerShell

About the Author

Ed Wilson is a senior consultant at Microsoft and a well-known scripting expert who delivers popular workshops. He's written several books on Windows scripting, including *Windows PowerShell Scripting Guide* and *Windows PowerShell 2.0 Best Practices*.

DEVELOPER ROADMAP

Start Here!
- Beginner-level instruction
- Easy-to-follow explanations and examples
- Exercises to build your first projects

Step by Step
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook

Developer Reference
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples

Focused Topics
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples

Microsoft®

Microsoft

Windows PowerShell™ 3.0 Step by Step

Ed Wilson

ISBN: 978-0-735-66339-8

2 3 4 5 6 7 8 9 10 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*.

Acquisitions and Developmental Editor: Michael Bolinger

Production Editor: Kristen Borg

Editorial Production: Zyg Group, LLC

Technical Reviewer: Thomas Lee

Copyeditor: Zyg Group, LLC

Indexer: Zyg Group, LLC

Cover Design: Twist Creative • Seattle

Cover Composition: Zyg Group, LLC

Illustrators: Rebecca Demarest and Robert Romano

[2013-04-19]

To Teresa, who makes each day seem fresh with opportunity and new with excitement.

Contents at a Glance

	Foreword	*xix*
	Introduction	*xxi*
CHAPTER 1	**Overview of Windows PowerShell 3.0**	**1**
CHAPTER 2	**Using Windows PowerShell Cmdlets**	**23**
CHAPTER 3	**Understanding and Using PowerShell Providers**	**65**
CHAPTER 4	**Using PowerShell Remoting and Jobs**	**107**
CHAPTER 5	**Using PowerShell Scripts**	**131**
CHAPTER 6	**Working with Functions**	**171**
CHAPTER 7	**Creating Advanced Functions and Modules**	**209**
CHAPTER 8	**Using the Windows PowerShell ISE**	**251**
CHAPTER 9	**Working with Windows PowerShell Profiles**	**267**
CHAPTER 10	**Using WMI**	**283**
CHAPTER 11	**Querying WMI**	**307**
CHAPTER 12	**Remoting WMI**	**337**
CHAPTER 13	**Calling WMI Methods on WMI Classes**	**355**
CHAPTER 14	**Using the CIM Cmdlets**	**367**
CHAPTER 15	**Working with Active Directory**	**383**
CHAPTER 16	**Working with the AD DS Module**	**419**
CHAPTER 17	**Deploying Active Directory with Windows Server 2012**	**447**
CHAPTER 18	**Debugging Scripts**	**461**
CHAPTER 19	**Handling Errors**	**501**
CHAPTER 20	**Managing Exchange Server**	**539**
APPENDIX A	**Windows PowerShell Core Cmdlets**	**571**
APPENDIX B	**Windows PowerShell Module Coverage**	**579**
APPENDIX C	**Windows PowerShell Cmdlet Naming**	**583**
APPENDIX D	**Windows PowerShell FAQ**	**587**
APPENDIX E	**Useful WMI Classes**	**597**
APPENDIX F	**Basic Troubleshooting Tips**	**621**
APPENDIX G	**General PowerShell Scripting Guidelines**	**625**
	Index	*633*

Contents

Foreword . . . . . *xix*
Introduction . . . . . *xxi*

Chapter 1 Overview of Windows PowerShell 3.0 1

Understanding Windows PowerShell 1
Using cmdlets 3
Installing Windows PowerShell 3
Deploying Windows PowerShell to down-level operating systems 4
Using command-line utilities 5
Security issues with Windows PowerShell 6
Controlling execution of PowerShell cmdlets 7
Confirming actions 8
Suspending confirmation of cmdlets 9
Working with Windows PowerShell 10
Accessing Windows PowerShell 10
Configuring the Windows PowerShell console 11
Supplying options for cmdlets 12
Working with the help options 13
Exploring commands: step-by-step exercises 19
Chapter 1 quick reference 22

Chapter 2 Using Windows PowerShell Cmdlets 23

Understanding the basics of cmdlets 23

Using the *Get-ChildItem* cmdlet 24

Obtaining a directory listing 24

Formatting a directory listing using the *Format-List* cmdlet 26

Using the *Format-Wide* cmdlet 27

Formatting a directory listing using *Format-Table* 29

Formatting output with *Out-GridView* 31

Leveraging the power of *Get-Command* 36

Searching for cmdlets using wildcard characters 36

Using the *Get-Member* cmdlet 44

Using the *Get-Member* cmdlet to examine properties and methods 44

Using the *New-Object* cmdlet 50

Creating and Using the *wshShell* Object 50

Using the Show-Command cmdlet 52

Windows PowerShell cmdlet naming helps you learn 54

Windows PowerShell verb grouping 54

Windows PowerShell verb distribution 55

Creating a Windows PowerShell profile 57

Finding all aliases for a particular object 59

Working with cmdlets: step-by-step exercises 59

Chapter 2 quick reference 63

Chapter 3 Understanding and Using PowerShell Providers 65

Understanding PowerShell providers 65

Understanding the alias provider 66

Understanding the certificate provider 68

Understanding the environment provider 76

Understanding the filesystem provider 80

Understanding the function provider 85

Using the registry provider to manage the Windows registry 87
The two registry drives 87
Understanding the variable provider 97
Exploring PowerShell providers: step-by-step exercises 101
Chapter 3 quick reference 106

Chapter 4 Using PowerShell Remoting and Jobs 107

Understanding Windows PowerShell remoting 107
Classic remoting 107
WinRM 112
Using Windows PowerShell jobs 119
Using Windows PowerShell remoting: step-by-step exercises 127
Chapter 4 quick reference 130

Chapter 5 Using PowerShell Scripts 131

Why write Windows PowerShell scripts? 131
Scripting fundamentals 133
Running Windows PowerShell scripts 133
Enabling Windows PowerShell scripting support 134
Transitioning from command line to script 136
Running Windows PowerShell scripts 138
Understanding variables and constants 141
Use of constants 146
Using the *While* statement 147
Constructing the *While* statement in PowerShell 148
A practical example of using the *While* statement 150
Using special features of Windows PowerShell 150
Using the *Do...While* statement 151
Using the range operator 152
Operating over an array 152
Casting to ASCII values 152

Using the *Do...Until* statement 153
Comparing the PowerShell *Do...Until* statement with VBScript . . . 154
Using the Windows PowerShell *Do* statement 154
The *For* statement 156
Using the *For* statement 156
Using the *Foreach* statement 158
Exiting the *Foreach* statement early 159
The *If* statement 161
Using assignment and comparison operators 163
Evaluating multiple conditions 164
The *Switch* statement 164
Using the *Switch* statement 165
Controlling matching behavior 167
Creating multiple folders: step-by-step exercises 168
Chapter 5 quick reference 170

Chapter 6 Working with Functions 171

Understanding functions 171
Using functions to provide ease of code reuse 178
Including functions in the Windows PowerShell environment 180
Using dot-sourcing 180
Using dot-sourced functions 182
Adding help for functions 184
Using a *here-string* object for help 184
Using two input parameters 186
Using a type constraint in a function 190
Using more than two input parameters 192
Use of functions to encapsulate business logic 194
Use of functions to provide ease of modification 196
Understanding filters 201
Creating a function: step-by-step exercises 205
Chapter 6 quick reference 208

Chapter 7 Creating Advanced Functions and Modules 209

The *[cmdletbinding]* attribute 209
Easy verbose messages 210
Automatic parameter checks 211
Adding support for the *-whatif* parameter 214
Adding support for the *-confirm* parameter 215
Specifying the default parameter set 216

The *parameter* attribute 217
The *mandatory* parameter property 217
The *position* parameter property 218
The *ParameterSetName* parameter property 219
The *ValueFromPipeline* property 220
The *HelpMessage* property 221

Understanding modules 222

Locating and loading modules 222
Listing available modules 223
Loading modules 225

Installing modules 227
Creating a per-user Modules folder 227
Working with the *$modulePath* variable 230
Creating a module drive 232
Checking for module dependencies 234
Using a module from a share 237

Creating a module 238

Creating an advanced function: step-by-step exercises 245

Chapter 7 quick reference 249

Chapter 8 Using the Windows PowerShell ISE 251

Running the Windows PowerShell ISE 251
Navigating the Windows PowerShell ISE 252
Working with the script pane 254
Tab expansion and IntelliSense 256

Working with Windows PowerShell ISE snippets 257
Using Windows PowerShell ISE snippets to create code. 257
Creating new Windows PowerShell ISE snippets 259
Removing user-defined Windows PowerShell ISE snippets 261
Using the Commands add-on: step-by-step exercises. 262
Chapter 8 quick reference 265

Chapter 9 Working with Windows PowerShell Profiles 267

Six Different PowerShell profiles 267
Understanding the six different Windows PowerShell profiles . . .268
Examining the *$profile* variable 268
Determining whether a specific profile exists. 270
Creating a new profile. 270
Design considerations for profiles 271
Using one or more profiles. 273
Using the All Users, All Hosts profile 275
Using your own file 276
Grouping similar functionality into a module 277
Where to store the profile module 278
Creating a profile: step-by-step exercises. 278
Chapter 9 quick reference 282

Chapter 10 Using WMI 283

Understanding the WMI model 284
Working with objects and namespaces 284
Listing WMI providers 289
Working with WMI classes. 289
Querying WMI. 293
Obtaining service information: step-by-step exercises 298
Chapter 10 quick reference. 305

Chapter 11 Querying WMI **307**

Alternate ways to connect to WMI 307
Selective data from all instances 316
Selecting multiple properties. 316
Choosing specific instances 319
Utilizing an operator 321
Where is the *where*?. 325
Shortening the syntax 325
Working with software: step-by-step exercises. 327
Chapter 11 quick reference 335

Chapter 12 Remoting WMI **337**

Using WMI against remote systems 337
Supplying alternate credentials for the remote connection. 338
Using Windows PowerShell remoting to run WMI. 341
Using CIM classes to query WMI classes 343
Working with remote results. 344
Reducing data via Windows PowerShell parameters. 347
Running WMI jobs 350
Using Windows PowerShell remoting and WMI: Step-by-step exercises 352
Chapter 12 quick reference 354

Chapter 13 Calling WMI Methods on WMI Classes **355**

Using WMI cmdlets to execute instance methods 355
Using the *terminate* method directly 357
Using the *Invoke-WmiMethod* cmdlet 358
Using the *[wmi]* type accelerator 360
Using WMI to work with static methods. 361
Executing instance methods: step-by-step exercises 364
Chapter 13 quick reference 366

Chapter 14 Using the CIM Cmdlets 367

Using the CIM cmdlets to explore WMI classes 367
Using the *-classname* parameter 367
Finding WMI class methods 368
Filtering classes by qualifier 369
Retrieving WMI instances 371
Reducing returned properties and instances 372
Cleaning up output from the command 373
Working with associations 373
Retrieving WMI instances: step-by-step exercises 379
Chapter 14 quick reference 382

Chapter 15 Working with Active Directory 383

Creating objects in Active Directory 383
Creating an OU 383
ADSI providers 385
LDAP names 387
Creating users 393
What is user account control? 396
Working with users 397
Creating multiple organizational units: step-by-step exercises 412
Chapter 15 quick reference 418

Chapter 16 Working with the AD DS Module 419

Understanding the Active Directory module 419
Installing the Active Directory module 419
Getting started with the Active Directory module 421
Using the Active Directory module 421
Finding the FSMO role holders 422
Discovering Active Directory 428
Renaming Active Directory sites 431
Managing users 432
Creating a user 435
Finding and unlocking Active Directory user accounts 436

Finding disabled users 438
Finding unused user accounts 440
Updating Active Directory objects: step-by-step exercises 443
Chapter 16 quick reference 445

Chapter 17 Deploying Active Directory with Windows Server 2012 447

Using the Active Directory module to deploy a new forest 447
Adding a new domain controller to an existing domain 453
Adding a read-only domain controller 455
Domain controller prerequisites: step-by-step exercises 457
Chapter 17 quick reference 460

Chapter 18 Debugging Scripts 461

Understanding debugging in Windows PowerShell 461
Understanding three different types of errors 461
Using the *Set-PSDebug* cmdlet 467
Tracing the script 467
Stepping through the script 471
Enabling strict mode 479
Using *Set-PSDebug -Strict* *479*
Using the *Set-StrictMode* cmdlet 481
Debugging the script 483
Setting breakpoints 483
Setting a breakpoint on a line number 483
Setting a breakpoint on a variable 485
Setting a breakpoint on a command 489
Responding to breakpoints 490
Listing breakpoints 492
Enabling and disabling breakpoints 494
Deleting breakpoints 494
Debugging a function: step-by-step exercises 494
Chapter 18 quick reference 499

Chapter 19 Handling Errors **501**

Handling missing parameters 501
Creating a default value for a parameter 502
Making the parameter mandatory 503
Limiting choices 504
Using *PromptForChoice* to limit selections 504
Using *Test-Connection* to identify computer connectivity 506
Using the *-contains* operator to examine contents of an array . . . 507
Using the *-contains* operator to test for properties 509
Handling missing rights 512
Attempt and fail 512
Checking for rights and exiting gracefully 513
Handling missing WMI providers 513
Incorrect data types 523
Out-of-bounds errors 526
Using a boundary-checking function 526
Placing limits on the parameter 528
Using *Try...Catch...Finally* *529*
Catching multiple errors 532
Using *PromptForChoice* to limit selections: Step-by-step exercises 534
Chapter 19 quick reference 537

Chapter 20 Managing Exchange Server **539**

Exploring the Exchange 2010 cmdlets 539
Working with remote Exchange servers 540
Configuring recipient settings 544
Creating the user and the mailbox 544
Reporting user settings 548
Managing storage settings 550
Examining the mailbox database 550
Managing the mailbox database 551

Managing Exchange logging 553
Managing auditing 557
Parsing the audit XML file 562
Creating user accounts: step-by-step exercises 565
Chapter 20 quick reference 570

Appendix A Windows PowerShell Core Cmdlets 571

Appendix B Windows PowerShell Module Coverage 579

Appendix C Windows PowerShell Cmdlet Naming 583

Appendix D Windows PowerShell FAQ 587

Appendix E Useful WMI Classes 597

Appendix F Basic Troubleshooting Tips 621

Appendix G General PowerShell Scripting Guidelines 625

Index *633*

About the Author *667*

Foreword

I've always known that automation was a critical IT Pro skill. Automation dramatically increases both productivity and quality of IT operations; it is a transformational skill that improves both the companies and the careers of the individuals that master it. Improving IT Pro automation was my top priority when I joined Microsoft in 1999 as the Architect for management products and technologies. That led to inventing Windows PowerShell and the long hard road to making it a centerpiece of the Microsoft management story. Along the way, the industry made some dramatic shifts. These shifts make it even more critical for IT Pros to become experts of automation.

During the development of PowerShell V1, the team developed a very strong partnership with Exchange. We thought Exchange would drive industry adoption of PowerShell. You can imagine our surprise (and delight) when we discovered that the most active PowerShell V1 community was VMWare customers. I reached out to the VMWare team to find out why it was so successful with their customers. They explained to me that their customers were IT Pros that were barely keeping up with the servers they had. When they adopted virtualization, they suddenly had 5-10 times the number of servers so it was either "automate or drown." Their hair was on fire and PowerShell was a bucket of water.

The move to the cloud is another shift that increases the importance of automation. The entire DevOps movement is all about making change safe through changes in culture and automation. When you run cloud scale applications, you can't afford to have it all depend upon a smart guy with a cup of coffee and a mouse–you need to automate operations with scripts and workflows. When you read the failure reports of the biggest cloud outages, you see that the root cause is often manual configuration. When you have automation and an error occurs, you review the scripts and modify them to it doesn't happen again. With automation, Nietzsche was right: that which does not kill you strengthens you. It is no surprise that Azure has supported PowerShell for some time, but I was delighted to see that Amazon just released 587 cmdlets to manage AWS.

Learning automation with PowerShell is a critical IT Pro skill and there are few people better qualified to help you do that than Ed Wilson. Ed Wilson is the husband of The Scripting Wife and the man behind the wildly popular blog The Scripting Guy. It is no exaggeration to say that Ed and his wife Teresa are two of the most active people in the PowerShell community. Ed is known for his practical "how to" approach to PowerShell. Having worked with so many customers and people learning PowerShell, Ed knows what questions you are going to have even before you have them and has taken the time to lay it all out for you in his new book: Windows PowerShell 3.0 Step by Step.

—Jeffrey Snover, Distinguished Engineer and Lead Architect, Microsoft Windows

Introduction

Windows PowerShell 3.0 is an essential management and automation tool that brings the simplicity of the command line to next generation operating systems. Included in Windows 8 and Windows Server 2012, and portable to Windows 7 and Windows Server 2008 R2, Windows PowerShell 3.0 offers unprecedented power and flexibility to everyone from power users to enterprise network administrators and architects.

Who should read this book

This book exists to help IT Pros come up to speed quickly on the exciting Windows PowerShell 3.0 technology. *Windows PowerShell 3.0 Step by Step* is specifically aimed at several audiences, including:

- **Windows networking consultants** Anyone desiring to standardize and to automate the installation and configuration of dot-net networking components.
- **Windows network administrators** Anyone desiring to automate the day-to-day management of Windows dot-net networks.
- **Microsoft Certified Solutions Experts (MCSEs) and Microsoft Certified Trainers (MCTs)** Windows PowerShell is a key component of many Microsoft courses and certification exams.
- **General technical staff** Anyone desiring to collect information, configure settings on Windows machines.
- **Power users** Anyone wishing to obtain maximum power and configurability of their Windows machines either at home or in an unmanaged desktop workplace environment.

Assumptions

This book expects that you are familiar with the Windows operating system, and therefore basic networking terms are not explained in detail. The book does not expect you to have any background in programming, development, or scripting. All elements related to these topics, as they arise, are fully explained.

Who should not read this book

Not every book is aimed at every possible audience. This is not a Windows PowerShell 3.0 reference book, and therefore extremely deep, esoteric topics are not covered. While some advanced topics are covered, in general the discussion starts with beginner topics and proceeds through an intermediate depth. If you have never seen a computer, nor have any idea what a keyboard or a mouse are, then this book definitely is not for you.

Organization of this book

This book is divided into three sections, each of which focuses on a different aspect or technology within the Windows PowerShell world. The first section provides a quick overview of Windows PowerShell and its fundamental role in Windows Management. It then delves into the details of Windows PowerShell remoting. The second section covers the basics of Windows PowerShell scripting. The last portion of the book covers different management technology and discusses specific applications such as Active Directory and Exchange.

Finding your best starting point in this book

The different sections of *Windows PowerShell 3.0 Step by Step* cover a wide range of technologies associated with the data library. Depending on your needs and your existing understanding of Microsoft data tools, you may wish to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

If you are	Follow these steps
New to Windows PowerShell	Focus on Chapters 1–3 and 5–9, or read through the entire book in order.
An IT pro who knows the basics of Windows PowerShell and only needs to learn how to manage network resources	Briefly skim Chapters 1–3 if you need a refresher on the core concepts. Read up on the new technologies in Chapters 4 and 10–14.
Interested in Active Directory and Exchange	Read Chapters 15–17 and 20.
Interested in Windows PowerShell Scripting	Read Chapters 5–8, 18, and 19.

Most of the book's chapters include hands-on samples that let you try out the concepts just learned.

Conventions and features in this book

This book presents information using conventions designed to make the information readable and easy to follow.

- Each chapter concludes with two exercises.
- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.
- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (e.g. File | Close), means that you should select the first menu or menu item, then the next, and so on.

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- One of the following: Windows 7, Windows Server 2008 with Service Pack 2, Windows Server 2008 R2, Windows 8 or Windows Server 2012.
- Computer that has a 1.6GHz or faster processor (2GHz recommended)
- 1 GB (32 Bit) or 2 GB (64 Bit) RAM (Add 512 MB if running in a virtual machine or SQL Server Express Editions, more for advanced SQL Server editions)
- 3.5 GB of available hard disk space
- 5400 RPM hard disk drive
- DirectX 9 capable video card running at 1024 × 768 or higher-resolution display

- DVD-ROM drive (if installing Visual Studio from DVD)
- Internet connection to download software or chapter examples

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2010 and SQL Server 2008 products.

Code samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects, in both their pre-exercise and post-exercise formats, can be downloaded from the following page:

http://aka.ms/PowerShellSBS_book

Follow the instructions to download the scripts.zip file.

Note In addition to the code samples, your system should have Windows PowerShell 3.0 installed.

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. After you download the scripts.zip file, make sure you unblock it by right-clicking on the scripts.zip file, and then clicking on the Unblock button on the property sheet.
2. Unzip the scripts.zip file that you downloaded from the book's website (name a specific directory along with directions to create it, if necessary).

Acknowledgments

I'd like to thank the following people: my agent Claudette Moore, because without her this book would never have come to pass. My editors Devon Musgrave and Michael Bolinger for turning the book into something resembling English, and my technical

reviewer Thomas Lee whose attention to detail definitely ensured a much better book. Lastly I want to acknowledge my wife Teresa (aka the Scripting Wife) who read every page and made numerous suggestions that will be of great benefit to beginning scripters.

Errata and book support

We've 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://www.microsoftpressstore.com/title/ 9780735663398

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, 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's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 1

Overview of Windows PowerShell 3.0

After completing this chapter, you will be able to:

- Understand basic use and capabilities of Windows PowerShell.
- Install Windows PowerShell.
- Use basic command-line utilities inside Windows PowerShell.
- Use Windows PowerShell help.
- Run basic Windows PowerShell cmdlets.
- Get help on basic Windows PowerShell cmdlets.
- Configure Windows PowerShell to run scripts.

The release of Microsoft Windows PowerShell 3.0 marks a significant advance for the Windows network administrator. Combining the power of a full-fledged scripting language with access to command-line utilities, Windows Management Instrumentation (WMI), and even VBScript, Windows PowerShell provides the power and ease of use that have been missing from the Windows platform since the beginning of time. As part of the Microsoft Common Engineering Criteria, Windows PowerShell is quickly becoming the management solution for the Windows platform. IT professionals using the Windows Server 2012 core installation must come to grips with Windows PowerShell sooner rather than later.

Understanding Windows PowerShell

Perhaps the biggest obstacle for a Windows network administrator in migrating to Windows PowerShell 3.0 is understanding what PowerShell actually is. In some respects, it is a replacement for the venerable CMD (command) shell. In fact, on Windows Server 2012 running in core mode, it is possible to replace the CMD shell with Windows PowerShell so that when the server boots up, it uses Windows PowerShell as the interface. As shown here, after Windows PowerShell launches, you can use *cd* to change the working directory, and then use *dir* to produce a directory listing in exactly the same way you would perform these tasks from the CMD shell.

```
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

PS C:\Users\administrator> cd c:\
PS C:\> dir


    Directory: C:\


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         3/22/2012   4:03 AM            PerfLogs
d-r--         3/22/2012   4:24 AM            Program Files
d-r--         3/23/2012   6:02 PM            Users
d----         3/23/2012   4:59 PM            Windows
-a---         3/22/2012   4:33 AM         24 autoexec.bat
-a---         3/22/2012   4:33 AM         10 config.sys


PS C:\>
```

You can also combine traditional CMD interpreter commands with some of the newer utilities, such as *fsutil*. This is shown here:

```
PS C:\> md c:\test


    Directory: C:\


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         4/22/2012   5:01 PM            test


PS C:\> fsutil file createnew C:\test\mynewfile.txt 1000
File C:\test\mynewfile.txt is created
PS C:\> cd c:\test
PS C:\test> dir


    Directory: C:\test


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         4/22/2012   5:01 PM       1000 mynewfile.txt


PS C:\test>
```

The preceding two examples show Windows PowerShell being used in an interactive manner. Interactivity is one of the primary features of Windows PowerShell, and you can begin to use Windows PowerShell interactively by opening a Windows PowerShell prompt and typing commands. You can enter the commands one at a time, or you can group them together like a batch file. I will discuss this later because you will need more information to understand it.

Using cmdlets

In addition to using Windows console applications and built-in commands, you can also use the *cmdlets* (pronounced *commandlets*) that are built into Windows PowerShell. Cmdlets can be created by anyone. The Windows PowerShell team creates the core cmdlets, but many other teams at Microsoft were involved in creating the hundreds of cmdlets shipping with Windows 8. They are like executable programs, but they take advantage of the facilities built into Windows PowerShell, and therefore are easy to write. They are not scripts, which are uncompiled code, because they are built using the services of a special .NET Framework namespace. Windows PowerShell 3.0 comes with about 1,000 cmdlets on Windows 8, and as additional features and roles are added, so are additional cmdlets. These cmdlets are designed to assist the network administrator or consultant to leverage the power of Windows PowerShell without having to learn a scripting language. One of the strengths of Windows PowerShell is that cmdlets use a standard naming convention that follows a verb-noun pattern, such as *Get-Help*, *Get-EventLog,* or *Get-Process*. The cmdlets using the *get* verb display information about the item on the right side of the dash. The cmdlets that use the *set* verb modify or set information about the item on the right side of the dash. An example of a cmdlet that uses the *set* verb is *Set-Service*, which can be used to change the start mode of a service. All cmdlets use one of the standard verbs. To find all of the standard verbs, you can use the *Get-Verb* cmdlet. In Windows PowerShell 3.0, there are nearly 100 approved verbs.

Installing Windows PowerShell

Windows PowerShell 3.0 comes with Windows 8 Client and Windows Server 2012. You can download the Windows Management Framework 3.0 package containing updated versions of Windows Remote Management (WinRM), WMI, and Windows PowerShell 3.0 from the Microsoft Download center. Because Windows 8 and Windows Server 2012 come with Windows PowerShell 3.0, there is no Windows Management Framework 3.0 package available for download—it is not needed. In order to install Windows Management Framework 3.0 on Windows 7, Windows Server 2008 R2, and Windows Server 2008, they all must be running at least Service Pack (SP) 1 and the Microsoft .NET Framework 4.0. There is no package for Windows Vista, Windows Server 2003, or earlier versions of the operating system. You can run both Windows PowerShell 3.0 and Windows PowerShell 2.0 on the same system, but this requires both the .NET Framework 3.5 and 4.0.

To prevent frustration during the installation, it makes sense to use a script that checks for the operating system, service pack level, and .NET Framework 4.0. A sample script that will check for the prerequisites is Get-PowerShellRequirements.ps1, which follows.

```
Get-PowerShellRequirements.ps1
Param([string[]]$computer = @($env:computername, "LocalHost"))
 foreach ($c in $computer)
  {
    $o = Get-WmiObject win32_operatingsystem -cn $c
    switch ($o.version)
    {
        {$o.version -gt 6.2} {"$c is Windows 8 or greater"; break}
        {$o.version -gt 6.1}
          {
           If($o.ServicePackMajorVersion -gt 0){$sp = $true}
           If(Get-WmiObject Win32_Product -cn $c |
              where { $_.name -match '.NET Framework 4'}) {$net = $true }
           If($sp -AND $net) { "$c meets the requirements for PowerShell 3" ; break}
           ElseIF (!$sp) {"$c needs a service pack"; break}
           ELSEIF (!$net) {"$c needs a .NET Framework upgrade"} ; break}
        {$o.version -lt 6.1} {"$c does not meet standards for PowerShell 3.0"; break}
        Default {"Unable to tell if $c meets the standards for PowerShell 3.0"}
    }

  }
```

Deploying Windows PowerShell to down-level operating systems

After Windows PowerShell is downloaded from *http://www.microsoft.com/downloads*, you can deploy it to your enterprise by using any of the standard methods. Here are few of the methods that you can use to accomplish Windows PowerShell deployment:

- Create a Microsoft Systems Center Configuration Manager package and advertise it to the appropriate organizational unit (OU) or collection.
- Create a Group Policy Object (GPO) in Active Directory Domain Services (AD DS) and link it to the appropriate OU.
- Approve the update in Software Update Services (SUS) when available.
- Add the Windows Management Framework 3.0 packages to a central file share or webpage for self service.

If you are not deploying to an entire enterprise, perhaps the easiest way to install Windows PowerShell is to download the package and step through the wizard.

Note To use a command-line utility in Windows PowerShell, launch Windows PowerShell by choosing Start | Run | PowerShell. At the PowerShell prompt, type in the command to run.

Using command-line utilities

As mentioned earlier, command-line utilities can be used directly within Windows PowerShell. The advantages of using command-line utilities in Windows PowerShell, as opposed to simply running them in the CMD interpreter, are the Windows PowerShell pipelining and formatting features. Additionally, if you have batch files or CMD files that already use existing command-line utilities, you can easily modify them to run within the Windows PowerShell environment. The following procedure illustrates adding *ipconfig* commands to a text file.

Running *ipconfig* commands

1. Start Windows PowerShell by choosing Start | Run | Windows PowerShell. The PowerShell prompt will open by default at the root of your Documents folder.
2. Enter the command **ipconfig /all**. This is shown here:

```
PS C:\> ipconfig /all
```

3. Pipe the result of *ipconfig /all* to a text file. This is illustrated here:

```
PS C:\> ipconfig /all >ipconfig.txt
```

4. Open Notepad to view the contents of the text file, as follows:

```
PS C:\> notepad ipconfig.txt
```

Typing a single command into Windows PowerShell is useful, but at times you may need more than one command to provide troubleshooting information or configuration details to assist with setup issues or performance problems. This is where Windows PowerShell really shines. In the past, you would have either had to write a batch file or type the commands manually. This is shown in the TroubleShoot.bat script that follows.

TroubleShoot.bat

```
ipconfig /all >C:\tshoot.txt
route print >>C:\tshoot.txt
hostname >>C:\tshoot.txt
net statistics workstation >>C:\tshoot.txt
```

Of course, if you typed the commands manually, then you had to wait for each command to complete before entering the subsequent command. In that case, it was always possible to lose your place in the command sequence, or to have to wait for the result of each command. Windows PowerShell eliminates this problem. You can now enter multiple commands on a single line, and then leave the computer or perform other tasks while the computer produces the output. No batch file needs to be written to achieve this capability.

Tip Use multiple commands on a single Windows PowerShell line. Type each complete command, and then use a semicolon to separate each command.

The following exercise describes how to run multiple commands. The commands used in the procedure are in the RunningMultipleCommands.txt file.

Running multiple commands

1. Open Windows PowerShell by choosing Start | Run | Windows PowerShell. The PowerShell prompt will open by default at the root of your Documents And Settings folder.
2. Enter the **ipconfig /all** command. Pipe the output to a text file called Tshoot.txt by using the redirection arrow (>). This is the result:

   ```
   ipconfig /all >tshoot.txt
   ```

3. On the same line, use a semicolon to separate the *ipconfig /all* command from the *route print* command. Append the output from the command to a text file called Tshoot.txt by using the redirect-and-append arrow (>>). Here is the command so far:

   ```
   ipconfig /all >tshoot.txt; route print >>tshoot.txt
   ```

4. On the same line, use a semicolon to separate the *route print* command from the *hostname* command. Append the output from the command to a text file called Tshoot.txt by using the redirect-and-append arrow. The command up to this point is shown here:

   ```
   ipconfig /all >tshoot.txt; route print >>tshoot.txt; hostname >>tshoot
   .txt
   ```

5. On the same line, use a semicolon to separate the *hostname* command from the *net statistics workstation* command. Append the output from the command to a text file called Tshoot.txt by using the redirect-and-append arrow. The completed command looks like the following:

   ```
   ipconfig /all >tshoot.txt; route print >>tshoot.txt; netdiag /q >>tshoot
   .txt; net statistics workstation >>tshoot.txt
   ```

Security issues with Windows PowerShell

As with any tool as versatile as Windows PowerShell, there are bound to be some security concerns. Security, however, was one of the design goals in the development of Windows PowerShell.

When you launch Windows PowerShell, it opens in your Documents folder; this ensures you are in a directory where you will have permission to perform certain actions and activities. This is far safer than opening at the root of the drive, or even opening in system root.

To change to a directory in the Windows PowerShell console, you cannot automatically go up to the next level; you must explicitly name the destination of the change-directory operation (although you can use the *cd ..* command to move up one level).

The running of scripts is disabled by default and can be easily managed through group policy. It can also be managed on a per-user or per-session basis.

Controlling execution of PowerShell cmdlets

Have you ever opened a CMD interpreter prompt, typed in a command, and pressed Enter so that you could see what it does? What if that command happened to be *Format C:*? Are you sure you want to format your C drive? This section will cover some arguments that can be supplied to cmdlets that allow you to control the way they execute. Although not all cmdlets support these arguments, most of those included with Windows PowerShell do. The three arguments you can use to control execution are *-whatif*, *-confirm*, and suspend. *Suspend* is not really an argument that is supplied to a cmdlet, but rather is an action you can take at a confirmation prompt, and is therefore another method of controlling execution.

Note To use *-whatif* at a Windows PowerShell prompt, enter the cmdlet. Type the **-whatif** parameter after the cmdlet. This only works for cmdlets that change system state. Therefore, there is no *-whatif* parameter for cmdlets like *Get-Process* that only display information.

Windows PowerShell cmdlets that change system state (such as *Set-Service*) support a *prototype mode* that you can enter by using the *-whatif* parameter. The developer decides to implement *-whatif* when developing the cmdlet; however, the Windows PowerShell team recommends that developers implement *-whatif*. The use of the *-whatif* argument is shown in the following procedure. The commands used in the procedure are in the UsingWhatif.txt file.

Using *-whatif* to prototype a command

1. Open Windows PowerShell by choosing Start | Run | Windows PowerShell. The PowerShell prompt will open by default at the root of your Documents And Settings folder.
2. Start an instance of Notepad.exe. Do this by typing **notepad** and pressing the Enter key. This is shown here:

```
notepad
```

3. Identify the Notepad process you just started by using the *Get-Process* cmdlet. Type enough of the process name to identify it, and then use a wildcard asterisk (*) to avoid typing the entire name of the process, as follows:

```
Get-Process note*
```

4. Examine the output from the *Get-Process* cmdlet and identify the process ID. The output on my machine is shown here. Please note that in all likelihood, the process ID used by your instance of Notepad.exe will be different from the one on my machine.

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   1056 notepad
```

5. Use *-whatif* to see what would happen if you used *Stop-Process* to stop the process ID you obtained in step 4. This process ID will be found under the Id column in your output. Use the *-id* parameter to identify the Notepad.exe process. The command is as follows:

```
Stop-Process -id 1056 -whatif
```

6. Examine the output from the command. It tells you that the command will stop the Notepad process with the process ID that you used in your command.

```
What if: Performing operation "Stop-Process" on Target "notepad (1056)"
```

Confirming actions

As described in the previous section, you can use *-whatif* to prototype a cmdlet in Windows PowerShell. This is useful for seeing what a cmdlet would do; however, if you want to be prompted before the execution of the cmdlet, you can use the *-confirm* argument. The cmdlets used in the "Confirming the execution of cmdlets" procedure are listed in the ConfirmingExecutionOfCmdlets.txt file.

Confirming the execution of cmdlets

1. Open Windows PowerShell, start an instance of Notepad.exe, identify the process, and examine the output, just as in steps 1 through 4 in the previous exercise.

2. Use the *-confirm* argument to force a prompt when using the *Stop-Process* cmdlet to stop the Notepad process identified by the *Get-Process note** command. This is shown here:

```
Stop-Process -id 1768 -confirm
```

   The *Stop-Process* cmdlet, when used with the *-confirm* argument, displays the following confirmation prompt:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (1768)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

3. Type **y** and press Enter. The Notepad.exe process ends. The Windows PowerShell prompt returns to the default, ready for new commands, as shown here:

```
PS C:\>
```

Tip To suspend cmdlet confirmation, at the confirmation prompt from the cmdlet, type **s** and press Enter.

Suspending confirmation of cmdlets

The ability to prompt for confirmation of the execution of a cmdlet is extremely useful and at times may be vital to assisting in maintaining a high level of system uptime. There may be times when you type in a long command and then remember that you need to check on something else first. For example, you may be in the middle of stopping a number of processes, but you need to view details on the processes to ensure you do not stop the wrong one. For such eventualities, you can tell the confirmation you would like to suspend execution of the command. The commands used for suspending execution of a cmdlet are in the SuspendConfirmationOfCmdlets.txt file.

Suspending execution of a cmdlet

1. Open Windows PowerShell, start an instance of Notepad.exe, identify the process, and examine the output, just as in steps 1 through 4 in the previous exercise. The output on my machine is shown following. Please note that in all likelihood, the process ID used by your instance of Notepad.exe will be different from the one on my machine.

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   3576 notepad
```

2. Use the *-confirm* argument to force a prompt when using the *Stop-Process* cmdlet to stop the Notepad process identified by the *Get-Process note** command. This is illustrated here:

```
Stop-Process -id 3576 -confirm
```

   The *Stop-Process* cmdlet, when used with the *-confirm* argument, displays the following confirmation prompt:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (3576)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

3. To suspend execution of the *Stop-Process* cmdlet, enter **s**. A triple-arrow prompt will appear, as follows:

```
PS C:\>>>
```

4. Use the *Get-Process* cmdlet to obtain a list of all the running processes that begin with the letter *n*. The syntax is as follows:

```
Get-Process n*
```

On my machine, two processes appear. The Notepad process I launched earlier and another process. This is shown here:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   3576 notepad
     75       2     1776       2708    23     0.09    632 nvsvc32
```

5. Return to the previous confirmation prompt by typing **exit**.

Once again, the confirmation prompt appears as follows:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (3576)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

6. Type **y** and press Enter to stop the Notepad process. There is no further confirmation. The prompt now displays the default Windows PowerShell prompt, as shown here:

```
PS C:\>
```

Working with Windows PowerShell

This section will go into detail about how to access Windows PowerShell and configure the Windows PowerShell console.

Accessing Windows PowerShell

After Windows PowerShell is installed on a down-level system, it becomes available for immediate use. However, using the Windows flag key on the keyboard and pressing R to bring up a *run* command prompt—or mousing around and choosing Start | Run | Windows PowerShell all the time—will become time-consuming and tedious. (This is not quite as big a problem on Windows 8, where you can just type **PowerShell** on the Start screen). On Windows 8, I pin both Windows PowerShell and the PowerShell ISE to both the Start screen and the taskbar. On Windows Server 2012 in core mode, I replace the CMD prompt with the Windows PowerShell console. For me and the way I work, this is ideal, so I wrote a script to do it. This script can be called through a log-on script to automatically deploy the shortcut on the desktop. On Windows 8, the script adds both the Windows PowerShell ISE and the Windows PowerShell console to both the Start screen and the taskbar. On Windows 7, it adds both to the taskbar and to the Start menu. The script only works for U.S. English–language operating

systems. To make it work in other languages, change the value of *$pinToStart* or *$pinToTaskBar* to the equivalent values in the target language.

Note Using Windows PowerShell scripts is covered in Chapter 5, "Using PowerShell Scripts." See that chapter for information about how the script works and how to actually run the script.

The script is called PinToStartAndTaskBar.ps1, and is as follows:

PinToStartAndTaskBar.ps1

```
$pinToStart = "Pin to Start"
$pinToTaskBar = "Pin to Taskbar"
$file = @((Join-Path -Path $PSHOME  -childpath "PowerShell.exe"),
          (Join-Path -Path $PSHOME  -childpath "powershell_ise.exe") )
Foreach($f in $file)
 {$path = Split-Path $f
  $shell=New-Object -com "Shell.Application"
  $folder=$shell.Namespace($path)
  $item = $folder.parsename((Split-Path $f -leaf))
  $verbs = $item.verbs()
  foreach($v in $verbs)
    {if($v.Name.Replace("&","") -match $pinToStart){$v.DoIt()}}
  foreach($v in $verbs)
    {if($v.Name.Replace("&","") -match $pinToTaskBar){$v.DoIt()}} }
```

Configuring the Windows PowerShell console

Many items can be configured for Windows PowerShell. These items can be stored in a Psconsole file. To export the console configuration file, use the *Export-Console* cmdlet, as shown here:

```
PS C:\> Export-Console myconsole
```

The Psconsole file is saved in the current directory by default and has an extension of .psc1. The Psconsole file is saved in XML format. A generic console file is shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>3.0</PSVersion>
  <PSSnapIns />
</PSConsoleFile>
```

Controlling PowerShell launch options

1. Launch Windows PowerShell without the banner by using the *-nologo* argument. This is shown here:

   ```
   PowerShell -nologo
   ```

2. Launch a specific version of Windows PowerShell by using the *-version* argument. (To launch Windows PowerShell 2.0, you must install the .NET Framework 3.5). This is shown here:

```
PowerShell -version 2
```

3. Launch Windows PowerShell using a specific configuration file by specifying the *-psconsolefile* argument, as follows:

```
PowerShell -psconsolefile myconsole.psc1
```

4. Launch Windows PowerShell, execute a specific command, and then exit by using the *-command* argument. The command itself must be prefixed by an ampersand (&) and enclosed in curly brackets. This is shown here:

```
Powershell -command "& {Get-Process}"
```

Supplying options for cmdlets

One of the useful features of Windows PowerShell is the standardization of the syntax in working with cmdlets. This vastly simplifies the learning of the new shell and language. Table 1-1 lists the common parameters. Keep in mind that some cmdlets cannot implement some of these parameters. However, if these parameters are used, they will be interpreted in the same manner for all cmdlets, because the Windows PowerShell engine itself interprets the parameters.

TABLE 1-1 Common parameters

Parameter	Meaning
-whatif	Tells the cmdlet to not execute, but to tell you what would happen if the cmdlet were to run.
-confirm	Tells the cmdlet to prompt before executing the command.
-verbose	Instructs the cmdlet to provide a higher level of detail than a cmdlet not using the verbose parameter.
-debug	Instructs the cmdlet to provide debugging information.
-ErrorAction	Instructs the cmdlet to perform a certain action when an error occurs. Allowed actions are *continue*, *stop*, *silentlyContinue*, and *inquire*.
-ErrorVariable	Instructs the cmdlet to use a specific variable to hold error information. This is in addition to the standard $error variable.
-OutVariable	Instructs the cmdlet to use a specific variable to hold the output information.
-OutBuffer	Instructs the cmdlet to hold a certain number of objects before calling the next cmdlet in the pipeline.

Note To get help on any cmdlet, use the *Get-Help <cmdletname>* cmdlet. For example, use *Get-Help Get-Process* to obtain help with using the *Get-Process* cmdlet.

Working with the help options

One of the first commands to run when opening Windows PowerShell for the first time is the *Update-Help* cmdlet. This is because Windows PowerShell does not ship help files with the product. This does not mean that no help presents itself—it does mean that help beyond simple syntax display requires an additional download.

A default installation of Windows PowerShell 3.0 contains numerous modules that vary from installation to installation depending upon the operating system features and roles selected. In fact, Windows PowerShell 3.0 installed on Windows 7 workstations contains far fewer modules and cmdlets than are available on a similar Windows 8 workstation. This does not mean all is chaos, however, because the essential Windows PowerShell cmdlets—the *core* cmdlets—remain unchanged from installation to installation. The difference between installations is because additional features and roles often install additional Windows PowerShell modules and cmdlets.

The modular nature of Windows PowerShell requires additional consideration when updating help. Simply running *Update-Help* does not update all of the modules loaded on a particular system. In fact, some modules may not support updatable help at all—these generate an error when you attempt to update help. The easiest way to ensure you update all possible help is to use both the *module* parameter and the *force* switched parameter. The command to update help for all installed modules (that support updatable help) is shown here:

```
Update-Help -Module * -Force
```

The result of running the *Update-Help* cmdlet on a typical Windows 8 client system is shown in Figure 1-1.

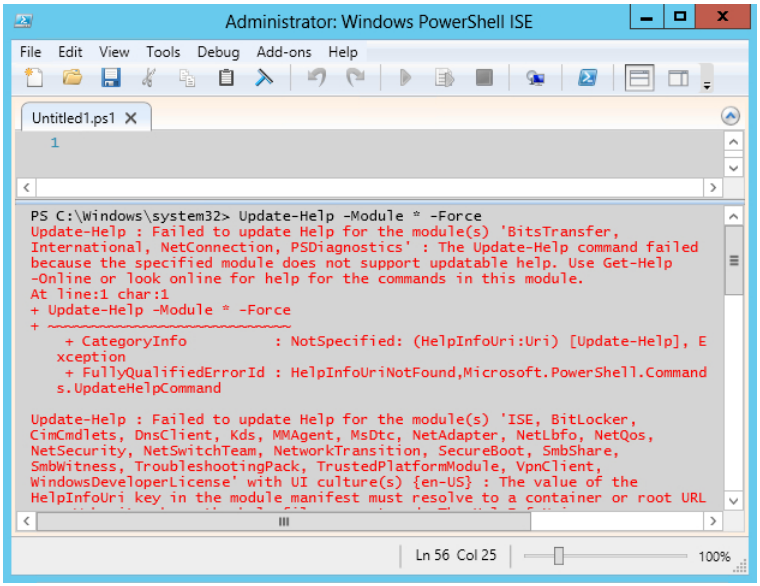

FIGURE 1-1 Errors appear when attempting to update help files that do not support updatable help.

One way to update help and not to receive a screen full of error messages is to run the *Update-Help* cmdlet and suppress the errors all together. This technique appears here:

```
Update-Help -Module * -Force -ea 0
```

The problem with this approach is that you can never be certain that you have actually received updated help for everything you wanted to update. A better approach is to hide the errors during the update process, but also to display errors after the update completes. The advantage to this approach is the ability to display cleaner errors. The UpdateHelpTrackErrors.ps1 script illustrates this technique. The first thing the UpdateHelpTrackErrors.ps1 script does is to empty the error stack by calling the *clear* method. Next, it calls the *Update-Help* module with both the *module* parameter and the *force* switched parameter. In addition, it uses the *ErrorAction* parameter (*ea* is an alias for this parameter) with a value of 0. A 0 value means that errors will not be displayed when the command runs. The script concludes by using a *For* loop to walk through the errors and displays the error exceptions. The complete UpdateHelpTrackErrors.ps1 script appears here.

Note For information about writing Windows PowerShell scripts and about using the *For* loop, see Chapter 5.

UpdateHelpTrackErrors.ps1

```
$error.Clear()
Update-Help -Module * -Force -ea 0
For ($i = 0 ; $i -lt $error.Count ; $i ++)
  { "`nerror $i" ; $error[$i].exception }
```

Once the UpdateHelpTrackErrors script runs, a progress bar displays indicating the progress as the updatable help files update. Once the script completes, any errors appear in order. The script and associated errors appear in Figure 1-2.

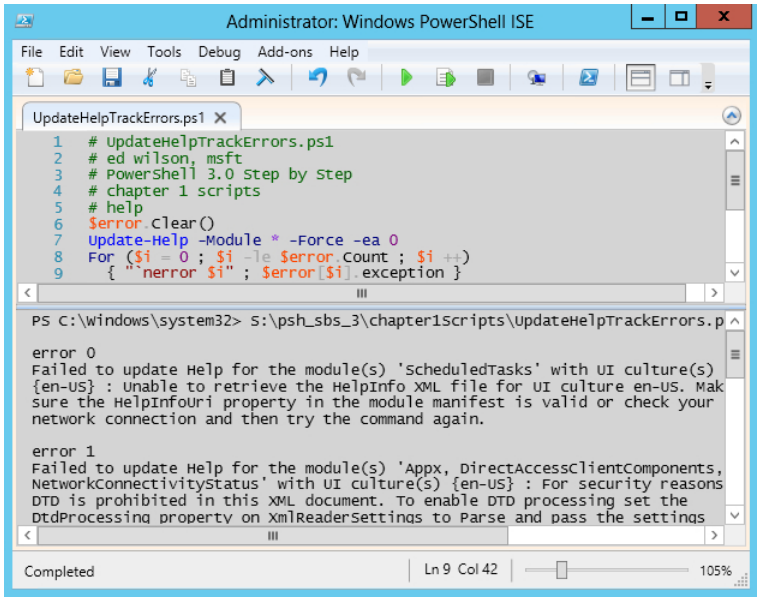

FIGURE 1-2 Cleaner error output from updatable help generated by the UpdateHelpTrackErrors script.

You can also determine which modules receive updated help by running the *Update-Help* cmdlet with the *-verbose* parameter. Unfortunately, when you do this, the output scrolls by so fast that it is hard to see what has actually updated. To solve this problem, redirect the verbose output to a text file. In the command that follows, all modules attempt to update *help*. The verbose messages redirect to a text file named updatedhelp.txt in a folder named fso off the root.

```
Update-Help -module * -force -verbose 4>>c:\fso\updatedhelp.txt
```

Windows PowerShell has a high level of discoverability; that is, to learn how to use PowerShell, you can simply use PowerShell. Online help serves an important role in assisting in this discoverability. The help system in Windows PowerShell can be entered by several methods. To learn about using Windows PowerShell, use the *Get-Help* cmdlet as follows:

```
Get-Help Get-Help
```

This command prints out help about the *Get-Help* cmdlet. The output from this cmdlet is illustrated here:

```
NAME
    Get-Help

SYNOPSIS
    Displays information about Windows PowerShell commands and concepts.


SYNTAX
    Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>] [-Full
    [<SwitchParameter>]] [-Functionality <String>] [-Path <String>] [-Role
    <String>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>]
    [-Functionality <String>] [-Path <String>] [-Role <String>] -Detailed
    [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>]
    [-Functionality <String>] [-Path <String>] [-Role <String>] -Examples
    [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>]
    [-Functionality <String>] [-Path <String>] [-Role <String>] -Online
    [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>]
    [-Functionality <String>] [-Path <String>] [-Role <String>] -Parameter <String>
    [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>]
    [-Functionality <String>] [-Path <String>] [-Role <String>] -ShowWindow
    [<SwitchParameter>] [<CommonParameters>]


DESCRIPTION
    The Get-Help cmdlet displays information about Windows PowerShell concepts and
    commands, including cmdlets, providers, functions, aliases and scripts.

    Get-Help gets the help content that it displays from help files on your
    computer. Without the help files, Get-Help displays only basic information
    about commands. Some Windows PowerShell modules come with help files. However,
    beginning in Windows PowerShell 3.0, the modules that come with Windows
    PowerShell do not include help files. To download or update the help files for
    a module in Windows PowerShell 3.0, use the Update-Help cmdlet. You can also
    view the help topics for Windows PowerShell online in the TechNet Library at http:
    //go.microsoft.com/fwlink/?LinkID=107116

    To get help for a Windows PowerShell command, type "Get-Help" followed by the
    command name. To get a list of all help topics on your system, type "Get-Help
    *".

    Conceptual help topics in Windows PowerShell begin with "about_", such as
    "about_Comparison_Operators". To see all "about_" topics, type "Get-Help
    about_*". To see a particular topic, type "Get-Help about_<topic-name>", such
    as "Get-Help about_Comparison_Operators".
```

```
    You can display the entire help topic or use the parameters of the Get-Help
    cmdlet to get selected parts of the topic, such as the syntax, parameters, or
    examples. You can also use the Online parameter to display an online version of
    a help topic for a command in your Internet browser.

    If you type "Get-Help" followed by the exact name of a help topic, or by a word
    unique to a help topic, Get-Help displays the topic contents. If you enter a
    word or word pattern that appears in several help topic titles, Get-Help
    displays a list of the matching titles. If you enter a word that does not
    appear in any help topic titles, Get-Help displays a list of topics that
    include that word in their contents.

    In addition to "Get-Help", you can also type "help" or "man", which displays
    one screen of text at a time, or "<cmdlet-name> -?", which is identical to
    Get-Help but works only for cmdlets.

    For information about the symbols that Get-Help displays in the command syntax
    diagram, see about_Command_Syntax http://go.microsoft.com/fwlink/?LinkID=113215.
    For information about parameter attributes, such as Required and Position, see
    about_Parameters http://go.microsoft.com/fwlink/?LinkID=113243.


RELATED LINKS
    Online Version: http://go.microsoft.com/fwlink/?LinkID=113316
    Get-Command
    Get-Member
    Get-PSDrive
    about_Command_Syntax
    about_Comment_Based_Help
    about_Parameters

REMARKS
    To see the examples, type: "Get-Help Get-Help -examples".
    For more information, type: "Get-Help Get-Help -detailed".
    For technical information, type: "Get-Help Get-Help -full".
    For online help, type: "Get-Help Get-Help -online"
```

The good thing about help with the Windows PowerShell is that it not only displays help about cmdlets, which you would expect, but it also has three levels of display: normal, detailed, and full. Additionally, you can obtain help about concepts in Windows PowerShell. This last feature is equivalent to having an online instruction manual. To retrieve a listing of all the conceptual help articles, use the *Get-Help about** command, as follows:

```
Get-Help about*
```

Suppose you do not remember the exact name of the cmdlet you wish to use, but you remember it was a *get* cmdlet? You can use a wildcard, such as an asterisk (*), to obtain the name of the cmdlet. This is shown here:

```
Get-Help get*
```

This technique of using a wildcard operator can be extended further. If you remember that the cmdlet was a *get* cmdlet, and that it started with the letter *p*, you can use the following syntax to retrieve the desired cmdlet:

```
Get-Help get-p*
```

Suppose, however, that you know the exact name of the cmdlet, but you cannot exactly remember the syntax. For this scenario, you can use the *-examples* argument. For example, for the *Get-PSDrive* cmdlet, you would use *Get-Help* with the *-examples* argument, as follows:

```
Get-Help Get-PSDrive -examples
```

To see help displayed one page at a time, you can use the *Help* function. The *Help* function passes your input to the *Get-Help* cmdlet, and pipelines the resulting information to the *more.com* utility. This causes output to display one page at a time in the Windows PowerShell console. This is useful if you want to avoid scrolling up and down to see the help output.

Note Keep in mind that in the Windows PowerShell ISE, the pager does not work, and therefore you will see no difference in output between *Get-Help* and *Help*. In the ISE, both *Get-Help* and *Help* behave the same way. However, it is likely that if you are using the Windows PowerShell ISE, you will use *Show-Command* for your help instead of relying on *Get-Help*.

This formatted output is shown in Figure 1-3.

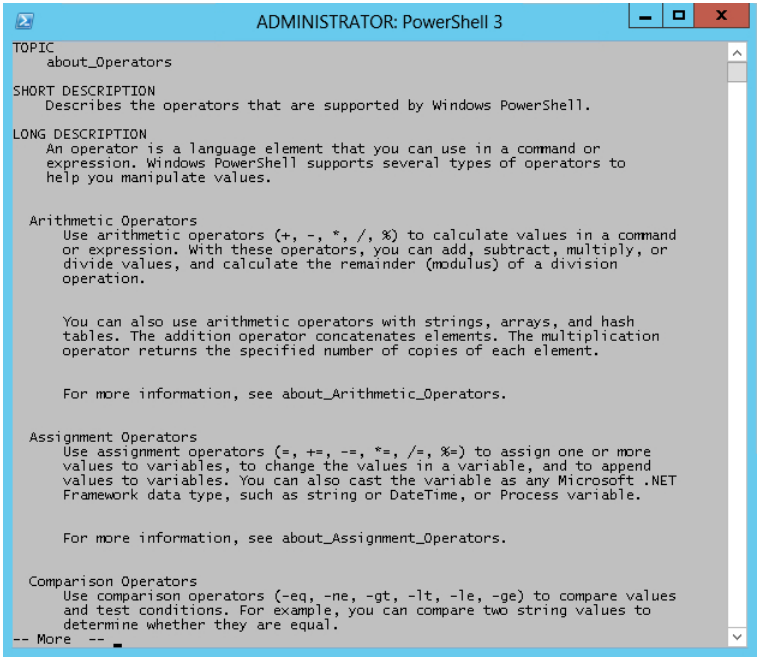

FIGURE 1-3 Using *Help* to display information one page at a time.

Getting tired of typing *Get-Help* all the time? After all, it is eight characters long. The solution is to create an alias to the *Get-Help* cmdlet. An alias is a shortcut keystroke combination that will launch a program or cmdlet when typed. In the "Creating an alias for the *Get-Help* cmdlet" procedure, you will assign the *Get-Help* cmdlet to the G+H key combination.

Note When creating an alias for a cmdlet, confirm it does not already have an alias by using *Get-Alias*. Use *New-Alias* to assign the cmdlet to a unique keystroke combination.

Creating an alias for the *Get-Help* cmdlet

1. Open Windows PowerShell by choosing Start | Run | Windows PowerShell. The PowerShell prompt will open by default at the root of your Documents folder.
2. Retrieve an alphabetic listing of all currently defined aliases, and inspect the list for one assigned to either the *Get-Help* cmdlet or the keystroke combination G+H. The command to do this is as follows:

```
Get-Alias sort
```

3. After you have determined that there is no alias for the *Get-Help* cmdlet and that none is assigned to the G+H keystroke combination, review the syntax for the *New-Alias* cmdlet. Use the *-full* argument to the *Get-Help* cmdlet. This is shown here:

```
Get-Help New-Alias -full
```

4. Use the *New-Alias* cmdlet to assign the G+H keystroke combination to the *Get-Help* cmdlet. To do this, use the following command:

```
New-Alias gh Get-Help
```

Exploring commands: step-by-step exercises

In the following exercises, you'll explore the use of command-line utilities in Windows PowerShell. You will see that it is as easy to use command-line utilities in Windows PowerShell as in the CMD interpreter; however, by using such commands in Windows PowerShell, you gain access to new levels of functionality.

Using command-line utilities

1. Open Windows PowerShell by choosing Start | Run | Windows PowerShell. The PowerShell prompt will open by default at the root of your Documents folder.
2. Change to the C:\root directory by typing **cd c:** inside the PowerShell prompt:

   ```
   cd c:\
   ```

3. Obtain a listing of all the files in the C:\root directory by using the *dir* command:

   ```
   dir
   ```

4. Create a directory off the C:\root directory by using the *md* command:

   ```
   md mytest
   ```

5. Obtain a listing of all files and folders off the root that begin with the letter *m*:

   ```
   dir m*
   ```

6. Change the working directory to the PowerShell working directory. You can do this by using the *Set-Location* command as follows:

   ```
   Set-Location $pshome
   ```

7. Obtain a listing of memory counters related to the available bytes by using the *typeperf* command. This command is shown here:

   ```
   typeperf "\memory\available bytes"
   ```

8. After a few counters have been displayed in the PowerShell window, press Ctrl+C to break the listing.
9. Display the current boot configuration by using the *bootcfg* command (note that you must run this command with admin rights):

   ```
   bootcfg
   ```

10. Change the working directory back to the C:\Mytest directory you created earlier:

    ```
    Set-Location c:\mytest
    ```

11. Create a file named mytestfile.txt in the C:\Mytest directory. Use the *fsutil* utility, and make the file 1,000 bytes in size. To do this, use the following command:

    ```
    fsutil file createnew mytestfile.txt 1000
    ```

12. Obtain a directory listing of all the files in the C:\Mytest directory by using the *Get-ChildItem* cmdlet.
13. Print out the current date by using the *Get-Date* cmdlet.

14. Clear the screen by using the *cls* command.

15. Print out a listing of all the cmdlets built into Windows PowerShell. To do this, use the *Get-Command* cmdlet.

16. Use the *Get-Command* cmdlet to get the *Get-Alias* cmdlet. To do this, use the *-name* argument while supplying *Get-Alias* as the value for the argument. This is shown here:

```
Get-Command -name Get-Alias
```

    This concludes the step-by-step exercise. Exit Windows PowerShell by typing **exit** and pressing Enter.

In the following exercise, you'll use various help options to obtain assistance with various cmdlets.

Obtaining help

1. Open Windows PowerShell by choosing Start | Run | Windows PowerShell. The PowerShell prompt will open by default at the root of your Documents folder.

2. Use the *Get-Help* cmdlet to obtain help about the *Get-Help* cmdlet. Use the command *Get-Help Get-Help* as follows:

```
Get-Help Get-Help
```

3. To obtain detailed help about the *Get-Help* cmdlet, use the *-detailed* argument as follows:

```
Get-Help Get-Help -detailed
```

4. To retrieve technical information about the *Get-Help* cmdlet, use the *-full* argument. This is shown here:

```
Get-Help Get-Help -full
```

5. If you only want to obtain a listing of examples of command usage, use the *-examples* argument as follows:

```
Get-Help Get-Help -examples
```

6. Obtain a listing of all the informational help topics by using the *Get-Help* cmdlet and the *about* noun with the asterisk (*) wildcard operator. The code to do this is shown here:

```
Get-Help about*
```

7. Obtain a listing of all the help topics related to *get* cmdlets. To do this, use the *Get-Help* cmdlet, and specify the word *get* followed by the wildcard operator as follows:

```
Get-Help get*
```

8. Obtain a listing of all the help topics related to *set* cmdlets. To do this, use the *Get-Help* cmdlet, followed by the *set* verb, followed by the asterisk wildcard. This is shown here:

```
Get-Help set*
```

This concludes this exercise. Exit Windows PowerShell by typing **exit** and pressing Enter.

Chapter 1 quick reference

To	Do This
Use an external command-line utility	Type the name of the command-line utility while inside Windows PowerShell.
Use multiple external command-line utilities sequentially	Separate each command-line utility with a semicolon on a single Windows PowerShell line.
Obtain a list of running processes	Use the *Get-Process* cmdlet.
Stop a process	Use the *Stop-Process* cmdlet and specify either the name or the process ID as an argument.
Model the effect of a cmdlet before actually performing the requested action	Use the *-whatif* argument.
Instruct Windows PowerShell to start up, run a cmdlet, and then exit	Use the *PowerShell* command while prefixing the cmdlet with & and enclosing the name of the cmdlet in curly brackets.
Prompt for confirmation before stopping a process	Use the *Stop-Process* cmdlet while specifying the *-confirm* argument.

CHAPTER 4

Using PowerShell Remoting and Jobs

After completing this chapter, you will be able to:

- Use Windows PowerShell remoting to connect to a remote system.
- Use Windows PowerShell remoting to run commands on a remote system.
- Use Windows PowerShell jobs to run commands in the background.
- Receive the results of background jobs.
- Keep the results from background jobs.

Understanding Windows PowerShell remoting

One of the great improvements in Microsoft Windows PowerShell 3.0 is the change surrounding remoting. The configuration is easier than it was in Windows PowerShell 2.0, and in most cases, Windows PowerShell remoting just works. When talking 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 may not be appropriate.

Classic remoting

Classic remoting in Windows PowerShell relies on protocols such as DCOM and RPC to make connections to remote machines. Traditionally, these protocols require opening many ports in the firewall and starting various services that 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* parameter of the *Get-Help* cmdlet. This command produces a nice list of all cmdlets that have native support for remoting. The command and associated output appear here:

```
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.
Clear-EventLog                    Deletes all entries from specified event logs on
                                  the local or remote computers.
Connect-PSSession                 Reconnects to disconnected sessions.
Connect-WSMan                     Connects to the WinRM service on a remote
                                  computer.
Disconnect-PSSession              Disconnects from a session.
Disconnect-WSMan                  Disconnects the client from the WinRM service on
                                  a remote computer.
Enter-PSSession                   Starts an interactive session with a remote
                                  computer.
Get-CimAssociatedInstance
                                  Get-CimAssociatedInstance [-InputObject]
                                  <ciminstance> [[-Association] <string>]
                                  [-ResultClassName <string>] [-Namespace <string>]
                                  [-OperationTimeoutSec <uint32>] [-ResourceUri
                                  <uri>] [-ComputerName <string[]>] [-KeyOnly]
                                  [<CommonParameters>]

                                  Get-CimAssociatedInstance [-InputObject]
                                  <ciminstance> [[-Association] <string>]
                                  -CimSession <CimSession[]> [-ResultClassName
                                  <string>] [-Namespace <string>]
                                  [-OperationTimeoutSec <uint32>] [-ResourceUri
                                  <uri>] [-KeyOnly] [<CommonParameters>]

Get-CimClass
                                  Get-CimClass [[-ClassName] <string>]
                                  [[-Namespace] <string>] [-OperationTimeoutSec
                                  <uint32>] [-ComputerName <string[]>] [-MethodName
                                  <string>] [-PropertyName <string>]
                                  [-QualifierName <string>] [<CommonParameters>]

                                  Get-CimClass [[-ClassName] <string>]
                                  [[-Namespace] <string>] -CimSession
                                  <CimSession[]> [-OperationTimeoutSec <uint32>]
                                  [-MethodName <string>] [-PropertyName <string>]
                                  [-QualifierName <string>] [<CommonParameters>]

Write-EventLog                    Writes an event to an event log.
```

As you can see, many of the Windows PowerShell cmdlets that have the *-computername* parameter relate to Web Services Management (WSMAN), Common Information Model (CIM), or sessions. To remove these cmdlets from the list, modify the command a bit to use *Where-Object* (*?* Is an alias for *Where-Object*). The revised command and associated output appear here:

```
PS C:\> Get-Help * -Parameter computername -Category cmdlet | ? modulename -match
'PowerShell.Management' | sort name | ft name, synopsis -AutoSize -Wrap

Name              Synopsis
----              --------
Add-Computer      Add the local computer to a domain or workgroup.
Clear-EventLog    Deletes all entries from specified event logs on the local or
                  remote computers.
Get-EventLog      Gets the events in an event log, or a list of the event logs, on
                  the local or remote computers.
Get-HotFix        Gets the hotfixes that have been applied to the local and remote
                  computers.
Get-Process       Gets the processes that are running on the local computer or a
                  remote computer.
Get-Service       Gets the services on a local or remote computer.
Get-WmiObject     Gets instances of Windows Management Instrumentation (WMI)
                  classes or information about the available classes.
Invoke-WmiMethod  Calls Windows Management Instrumentation (WMI) methods.
Limit-EventLog    Sets the event log properties that limit the size of the event
                  log and the age of its entries.
New-EventLog      Creates a new event log and a new event source on a local or
                  remote computer.
Register-WmiEvent Subscribes to a Windows Management Instrumentation (WMI) event.
Remove-Computer   Removes the local computer from its domain.
Remove-EventLog   Deletes an event log or unregisters an event source.
Remove-WmiObject  Deletes an instance of an existing Windows Management
                  Instrumentation (WMI) class.
Rename-Computer   Renames a computer.
Restart-Computer  Restarts ("reboots") the operating system on local and remote
                  computers.
Set-Service       Starts, stops, and suspends a service, and changes its properties.
Set-WmiInstance   Creates or updates an instance of an existing Windows Management
                  Instrumentation (WMI) class.
Show-EventLog     Displays the event logs of the local or a remote computer in
                  Event Viewer.
Stop-Computer     Stops (shuts down) local and remote computers.
Test-Connection   Sends ICMP echo request packets ("pings") to one or more
                  computers.

<-- 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 to retrieve the data. Figure 4-1 displays the credential dialog box that appears when the cmdlet runs.

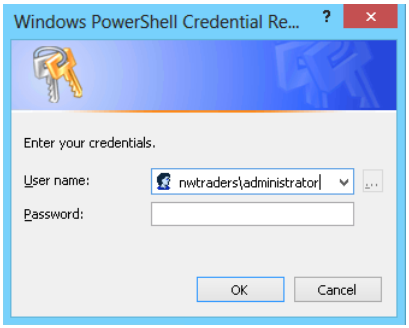

FIGURE 4-1 Cmdlets that support the *-credential* parameter prompt for credentials when supplied with a user name.

This technique of using the *-computername* and *-credential* parameters in a cmdlet appears here:

```
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 ADAccess                 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 don't have relaxed access rules. An example of this type of error appears here:

```
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* and *Get-Process*, do not have a *-credential* parameter, and therefore the commands associated with cmdlets such as *Get-Service* or *Get-Process* impersonate the logged-on user. Such a command appears here:

```
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 alternate credentials does not mean that the cmdlet must impersonate the logged-on user. Holding down the Shift key and right-clicking the Windows PowerShell icon from the taskbar brings up an action menu that allows you to run the program as a different user. This menu appears in Figure 4-2.

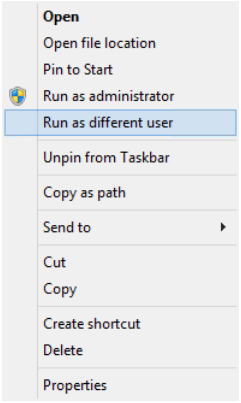

FIGURE 4-2 The menu from the Windows PowerShell console permits running with different security credentials.

The Run As Different User dialog box appears in Figure 4-3.

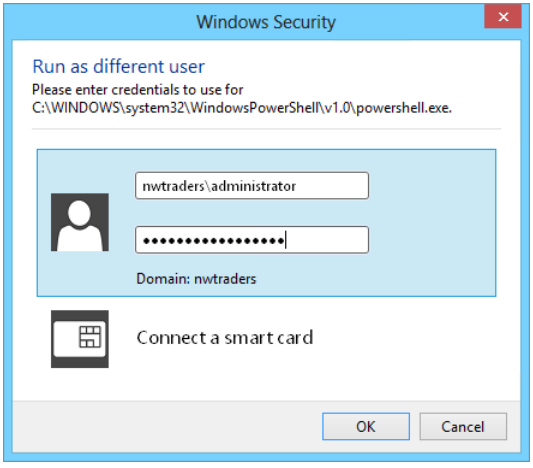

FIGURE 4-3 The Run As Different User dialog box permits entering a different user context.

Using the Run As Different User dialog box makes alternative credentials available for Windows PowerShell cmdlets that do not support the *-credential* parameter.

WinRM

Windows Server 2012 installs with Windows Remote Management (WinRM) configured and running to support remote Windows PowerShell commands. WinRM is Microsoft's 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 CIM cmdlets. 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. 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 Windows PowerShell remoting on the client machine. When running the *Enable-PSRemoting* function, the function performs the following steps:

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 WSMAN 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 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 syntax of this command appears here:

```
Enable-PSRemoting -force
```

The use of the *Enable-PSRemoting* function in interactive mode appears here, 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 machine.
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.powershell32 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 Windows PowerShell remoting is configured, use the *Test-WSMan* cmdlet to ensure that the WinRM remoting is properly configured and is accepting requests. A properly configured system replies with the information appearing here:

```
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. The output appearing here is from a domain controller running Windows 2008 with Windows PowerShell 2.0 installed and WinRM configured for remote access:

```
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
```

If WinRM is not configured, an error returns from the system. Such an error from a Windows 8 client appears here:

```
PS C:\> Test-WSMan -ComputerName w8c10
Test-WSMan : <f:WSManFault
xmlns:f="http://schemas.microsoft.com/wbem/wsman/1/wsmanfault" Code="2150859046"
Machine="w8c504.iammred.net"><f:Message>WinRM cannot complete the operation. Verify
that the specified computer name is valid, that the computer is accessible over the
network, and that a firewall exception for the WinRM service is enabled and allows
access from this computer. By default, the WinRM firewall exception for public
profiles limits access to remote computers within the same local subnet.
</f:Message></f:WSManFault>
At line:1 char:1
+ Test-WSMan -ComputerName w8c10
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (w8c10:String) [Test-WSMan], Invalid
   OperationException
    + FullyQualifiedErrorId : WsManError,Microsoft.WSMan.Management.TestWSManCommand
```

Keep in mind that configuring WinRM via the *Enable-PSRemoting* function does not enable the *Remote Management* firewall exception, and therefore *PING* commands will not work by default when pinging to a Windows 8 client system. This appears here:

```
PS C:\> ping w8c504

Pinging w8c504.iammred.net [192.168.0.56] with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.0.56:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss).
```

Pings to a Windows 2012 server, do however, work. This appears here:

```
PS C:\> ping w8s504

Pinging w8s504.iammred.net [192.168.0.57] with 32 bytes of data:
Reply from 192.168.0.57: bytes=32 time<1ms TTL=128
Reply from 192.168.0.57: bytes=32 time<1ms TTL=128
Reply from 192.168.0.57: bytes=32 time<1ms TTL=128
Reply from 192.168.0.57: bytes=32 time<1ms TTL=128
```

```
Ping statistics for 192.168.0.57:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Creating a remote Windows PowerShell session

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. This creates an interactive remote Windows PowerShell session on a target machine and uses the default remote endpoint. If you do not supply credentials, the remote session impersonates the currently logged on user The output appearing here illustrates connecting to a remote computer named dc1. Once the connection is established, the Windows PowerShell prompt changes to include the name of the remote system. *Set-Location* (which has an alias of *sl*) 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 prompt configured previously.

```
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:\>
```

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

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


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

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

Figure 4-4 displays a copy of the transcript from the previous session.

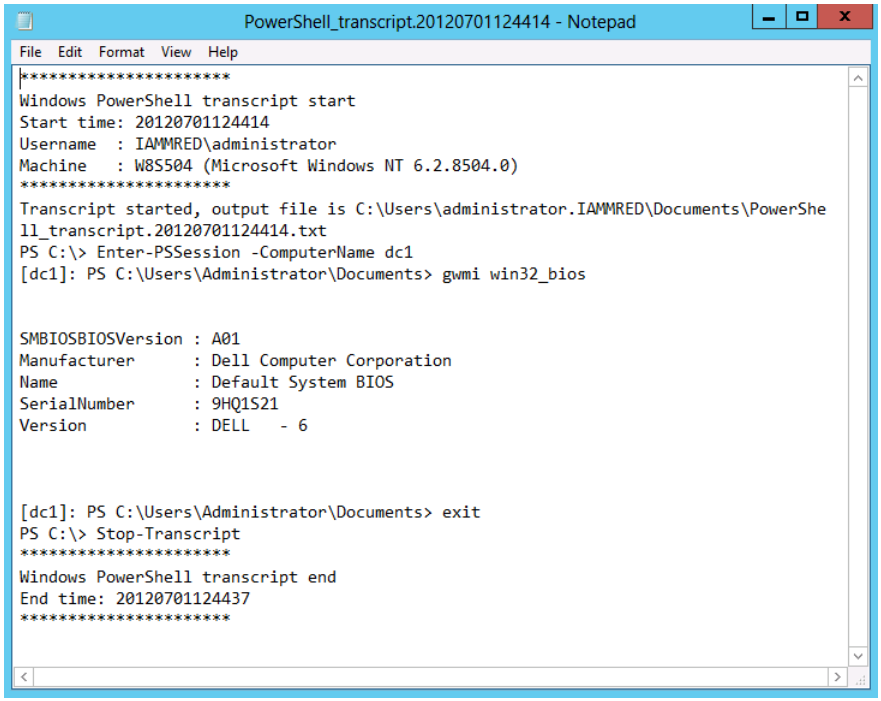

FIGURE 4-4 The Windows PowerShell transcript tool records commands and output received from a remote Windows PowerShell session.

If you anticipate making multiple connections to a remote system, use the *New-PSSession* cmdlet to create a remote Windows PowerShell session. *New-PSSession* permits you to store the remote session in a variable and provides you with the ability to enter and to 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 via 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 via the *exit* command. Next, the session is reentered, and the last process is retrieved. The session is exited once again. Finally, the *Get-PSSession* cmdlet retrieves Windows PowerShell sessions on the system, and all sessions are removed via 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:\>
```

Running a single Windows PowerShell command

If 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. You are still creating a remote session, but you are also removing the session. Therefore, if you have a lot of commands to run against the remote machine, a performance problem could arise. But for single commands, this technique works well. The technique is shown here, where the last process running on the Ex1 remote server appears:

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

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

If you have several commands, or if you anticipate making multiple connections, the *Invoke-Command* cmdlet accepts a session name or a session object in the same manner as the *Enter-PSSession* cmdlet. 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 the Windows PowerShell remote session is completed, the session stored in the *$dc1* variable is explicitly removed.

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

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


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

Using *Invoke-Command*, you can run 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 *PSCredential* object for the remote connections. Finally, the *Invoke-Command* cmdlet is used to make connections to all of 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 commands and associated output appear here:

```
PS C:\> $cn = "dc1","dc3","ex1","sql1","wsus1","wds1","hyperv1","hyperv2","hyperv3"
PS C:\> $cred = get-credential iammred\administrator
PS C:\> Invoke-Command -cn $cn -cred $cred -ScriptBlock {gwmi win32_bios}


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           : VRTUAL - 3000919
PSComputerName    : ex1


PS C:\>
```

Using Windows PowerShell jobs

Windows PowerShell jobs permit you to run one or more commands in the background. Once you start the Windows PowerShell job, the Windows PowerShell console returns immediately for further use. This permits you to accomplish multiple tasks at the same time. You can begin a new Windows

PowerShell job by using the *Start-Job* cmdlet. The command to run as a job is placed in a script block, and the jobs are sequentially named *Job1*, *Job2*, and so on. This is shown here:

```
PS C:\> Start-Job -ScriptBlock {get-process}

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
10     Job10           BackgroundJob   Running       True            localhost


PS C:\>
```

The jobs receive job IDs that are also sequentially numbered. The first job created in a Windows PowerShell console always has a job ID of 1. You can use either the job ID or the job name to obtain information about the job. This is shown here:

```
PS C:\> Get-Job -Name job10

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
10     Job10           BackgroundJob   Completed     True            localhost

PS C:\> Get-Job -Id 10

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
10     Job10           BackgroundJob   Completed     True            localhost


PS C:\>
```

Once you see that the job has completed, you can receive the job. The *Receive-Job* cmdlet returns the same information that returns if a job is not used. The Job1 output is shown here (truncated to save space):

```
PS C:\> Receive-Job -Name job10

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     62       9     1672       6032    80     0.00   1408 apdproxy
    132       9     2316       5632    62            1364 atieclxx
    122       7     1716       4232    32             948 atiesrxx
    114       9    14664      15372    48            1492 audiodg
    556      62    53928       5368   616     3.17   3408 CCC
     58       8     2960       7068    70     0.19    928 conhost
     32       5     1468       3468    52     0.00   5068 conhost
    784      14     3284       5092    56             416 csrss
    529      27     2928      17260   145             496 csrss
    182      13     8184      11152    96     0.50   2956 DCPSysMgr
    135      11     2880       7552    56            2056 DCPSysMgrSvc
 ... (truncated output)
```

Once a job has been received, that is it—the data is gone, unless you saved it to a variable or you call the *Receive-Job* cmdlet with the *-keep* switched parameter. The following code attempts to retrieve the information stored from job10, but as appears here, no data returns:

```
PS C:\> Receive-Job -Name job10
PS C:\>
```

What can be confusing about this is that the job still exists, and the *Get-Job* cmdlet continues to retrieve information about the job. This is shown here:

```
PS C:\> Get-Job -Id 10

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
10     Job10           BackgroundJob   Completed     False           localhost
```

As a best practice, use the *Remove-Job* cmdlet to delete remnants of completed jobs when you are finished using the job object. This will avoid confusion regarding active jobs, completed jobs, and jobs waiting to be processed. Once a job has been removed, the *Get-Job* cmdlet returns an error if you attempt to retrieve information about the job—because it no longer exists. This is illustrated here:

```
PS C:\> Remove-Job -Name job10
PS C:\> Get-Job -Id 10
Get-Job : The command cannot find a job with the job ID 10. Verify the value of the
Id parameter and then try the command again.
At line:1 char:1
+ Get-Job -Id 10
+ ~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (10:Int32) [Get-Job], PSArgumentException
    + FullyQualifiedErrorId : JobWithSpecifiedSessionNotFound,Microsoft.PowerShell.
   Commands.GetJobCommand
```

When working with the job cmdlets, I like to give the jobs their own name. A job that returns process objects via the *Get-Process* cmdlet might be called *getProc*. A contextual naming scheme works better than trying to keep track of names such as *Job1* and *Job2*. Do not worry about making your job names too long, because you can use wildcard characters to simplify the typing requirement. When you receive a job, make sure you store the returned objects in a variable. This is shown here:

```
PS C:\> Start-Job -Name getProc -ScriptBlock {get-process}

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
12     getProc         BackgroundJob   Running       True            localhost


PS C:\> Get-Job -Name get*

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
12     getProc         BackgroundJob   Completed     True            localhost
```

```
PS C:\> $procObj = Receive-Job -Name get*
PS C:\>
```

Once you have the returned objects in a variable, you can use the objects with other Windows PowerShell cmdlets. One thing to keep in mind is that the object is deserialized. This is shown here, where I use *gm* as an alias for the *Get-Member* cmdlet:

```
PS C:\> $procObj | gm


   TypeName: Deserialized.System.Diagnostics.Process
```

This means that not all the standard members from the *System.Diagnostics.Process* .NET Framework object are available. The default methods are shown here (*gps* is an alias for the *Get-Process* cmdlet, *gm* is an alias for *Get-Member*, and *-m* is enough of the *-membertype* parameter to distinguish it on the Windows PowerShell console line):

```
PS C:\> gps | gm -m method


   TypeName: System.Diagnostics.Process

Name                      MemberType Definition
----                      ---------- ----------
BeginErrorReadLine        Method     System.Void BeginErrorReadLine()
BeginOutputReadLine       Method     System.Void BeginOutputReadLine()
CancelErrorRead           Method     System.Void CancelErrorRead()
CancelOutputRead          Method     System.Void CancelOutputRead()
Close                     Method     System.Void Close()
CloseMainWindow           Method     bool CloseMainWindow()
CreateObjRef              Method     System.Runtime.Remoting.ObjRef CreateObjRef(type
                                     requestedType)
Dispose                   Method     System.Void Dispose()
Equals                    Method     bool Equals(System.Object obj)
GetHashCode               Method     int GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetType                   Method     type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
Kill                      Method     System.Void Kill()
Refresh                   Method     System.Void Refresh()
Start                     Method     bool Start()
ToString                  Method     string ToString()
WaitForExit               Method     bool WaitForExit(int milliseconds), System.Void
                                     WaitForExit()
WaitForInputIdle          Method     bool WaitForInputIdle(int milliseconds), bool
                                     WaitForInputIdle()
```

Methods from the deserialized object are shown here, where I use the same command I used previously:

```
PS C:\> $procObj | gm -m method


   TypeName: Deserialized.System.Diagnostics.Process

Name     MemberType Definition
----     ---------- ----------
ToString Method     string ToString(), string ToString(string format, System.IFormatProvider
formatProvider)


PS C:\>
```

A listing of the cmdlets that use the noun *job* is shown here:

```
PS C:\> Get-Command -Noun job | select name

Name
----
Get-Job
Receive-Job
Remove-Job
Resume-Job
Start-Job
Stop-Job
Suspend-Job
Wait-Job
```

When starting a Windows PowerShell job via the *Start-Job* cmdlet, you can specify a name to hold the returned job object. You can also assign the returned job object in a variable by using a straightforward value assignment. If you do both, you end up with two copies of the returned job object. This is shown here:

```
PS C:\> $rtn = Start-Job -Name net -ScriptBlock {Get-Net6to4Configuration}
PS C:\> Get-Job -Name net

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
18     net             BackgroundJob   Completed     True            localhost

PS C:\> $rtn

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
18     net             BackgroundJob   Completed     True            localhost
```

Retrieving the job via the *Receive-Job* cmdlet consumes the data. You cannot come back and retrieve the returned data again. This code shown here illustrates this concept:

```
PS C:\> Receive-Job $rtn


RunspaceId         : e8ed4ab6-eb88-478c-b2de-5991b5636ef1
Caption            :
Description        : 6to4 Configuration
ElementName        :
InstanceID         : ActiveStore
AutoSharing        : 0
PolicyStore        : ActiveStore
RelayName          : 6to4.ipv6.microsoft.com.
RelayState         : 0
ResolutionInterval : 1440
State              : 0



PS C:\> Receive-Job $rtn
PS C:\>
```

The next example illustrates examining the command and cleaning up the job. When you use *Receive-Job*, an error message is displayed. To find additional information about the code that triggered the error, use the job object stored in the *$rtn* variable or the *Get-Net6to4Configuration* job. You may prefer using the job object stored in the *$rtn* variable, as shown here:

```
PS C:\> $rtn.Command
Get-Net6to4Configuration
```

To clean up first, remove the leftover job objects by getting the jobs and removing the jobs. This is shown here:

```
PS C:\> Get-Job | Remove-Job
PS C:\> Get-Job
PS C:\>
```

When you create a new Windows PowerShell job, it runs in the background. There is no indication as the job runs whether it ends in an error or it's successful. Indeed, you do not have any way to tell when the job even completes, other than to use the *Get-Job* cmdlet several times to see when the job state changes from *running* to *completed*. For many jobs, this may be perfectly acceptable. In fact, it may even be preferable, if you wish to regain control of the Windows PowerShell console as soon as the job begins executing. On other occasions, you may wish to be notified when the Windows PowerShell job completes. To accomplish this, you can use the *Wait-Job* cmdlet. You need to give the *Wait-Job* cmdlet either a job name or a job ID. Once you have done this, the Windows PowerShell console will pause until the job completes. The job, with its *completed* status, displays on the console. You can then use the *Receive-Job* cmdlet to receive the deserialized objects and store them in a variable (*cn* is a parameter alias for the *-computername* parameter used in the *Get-WmiObject* command*)*. The command appearing here starts a job to receive software products installed on a remote server named hyperv1. It impersonates the currently logged-on user and stores the returned object in a variable named *$rtn*.

```
PS C:\> $rtn = Start-Job -ScriptBlock {gwmi win32_product -cn hyperv1}
PS C:\> $rtn

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
22     Job22           BackgroundJob   Running       True            localhost

PS C:\> Wait-Job -id 22

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
22     Job22           BackgroundJob   Completed     True            localhost

PS C:\> $prod = Receive-Job -id 22
PS C:\> $prod.Count
2
```

In a newly open Windows PowerShell console, the *Start-Job* cmdlet is used to start a new job. The returned job object is stored in the *$rtn* variable. You can pipeline the job object contained in the *$rtn* variable to the *Stop-Job* cmdlet to stop the execution of the job. If you try to use the job object in the *$rtn* variable directly to get job information, an error will be generated. This is shown here:

```
PS C:\> $rtn = Start-Job -ScriptBlock {gwmi win32_product -cn hyperv1}
PS C:\> $rtn | Stop-Job
PS C:\> Get-Job $rtn
Get-Job : The command cannot find the job because the job name
System.Management.Automation.PSRemotingJob was not found. Verify the value of the
Name parameter, and then try the command again.
At line:1 char:1
+ Get-Job $rtn
+ ~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (System.Manageme...n.PSRemotingJob:
   String) [Get-Job], PSArgumentException
    + FullyQualifiedErrorId : JobWithSpecifiedNameNotFound,Microsoft.PowerShell.
   Commands.GetJobCommand
```

You can pipeline the job object to the *Get-Job* cmdlet and see that the job is in a stopped state. Use the *Receive-Job* cmdlet to receive the job information and the *count* property to see how many software products are included in the variable, as shown here:

```
PS C:\> $rtn | Get-Job

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
2      Job2            BackgroundJob   Stopped       False           localhost


PS C:\> $products = Receive-Job -Id 2
PS C:\> $products.count
0
```

In the preceding list you can see that no software packages were enumerated. This is because the *Get-WmiObject* command to retrieve information from the *Win32_Product* class did not have time to finish.

If you want to keep the data from your job so that you can use it again later, and you do not want to bother storing it in an intermediate variable, use the *-keep* parameter. In the command that follows, the *Get-NetAdapter* cmdlet is used to return network adapter information.

```
PS C:\> Start-Job -ScriptBlock {Get-NetAdapter}

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
4      Job4            BackgroundJob   Running       True            localhost
```

When checking on the status of a background job, and you are monitoring a job you just created, use the *-newest* parameter instead of typing a job number, as it is easier to remember. This technique appears here:

```
PS C:\> Get-Job -Newest 1

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
4      Job4            BackgroundJob   Completed     True            localhost
```

Now, to retrieve the information from the job and to keep the information available, use the *-keep* switched parameter as illustrated here:

```
PS C:\> Receive-Job -Id 4 -Keep


ifAlias                                 : Ethernet
InterfaceAlias                          : Ethernet
ifIndex                                 : 12
ifDesc                                  : Microsoft Hyper-V Network Adapter
ifName                                  : Ethernet_7
DriverVersion                           : 6.2.8504.0
LinkLayerAddress                        : 00-15-5D-00-2D-07
MacAddress                              : 00-15-5D-00-2D-07
LinkSpeed                               : 10 Gbps
MediaType                               : 802.3
PhysicalMediaType                       : Unspecified
AdminStatus                             : Up
MediaConnectionState                    : Connected
DriverInformation                       : Driver Date 2006-06-21 Version
                                          6.2.8504.0 NDIS 6.30

DriverFileName                          : netvsc63.sys
NdisVersion                             : 6.30
ifOperStatus                            : Up
RunspaceId                              : 9ce8f8e6-1a09-4103-a508-c60398527
<output truncated>
```

You can continue to work directly with the output in a normal Windows PowerShell fashion, like so:

```
PS C:\> Receive-Job -Id 4 -Keep | select name

name
----
Ethernet


PS C:\> Receive-Job -Id 4 -Keep | select transmitlinksp*

                                                                TransmitLinkSpeed
                                                                -----------------
                                                                      10000000000
```

Using Windows PowerShell remoting: step-by-step exercises

In this exercise, you will practice using Windows PowerShell remoting to run remote commands. For the purpose of this exercise, you can use your local computer. First, you will open the Windows PowerShell console, supply alternate credentials, create a Windows PowerShell remote session, and run various commands. Next, you will create and receive Windows PowerShell jobs.

Supplying alternate credentials for remote Windows PowerShell sessions

1. Log on to your computer with a user account that does not have administrator rights.
2. Open the Windows PowerShell console.
3. Notice the Windows PowerShell console prompt. An example of such a prompt appears here:

   ```
   PS C:\Users\ed.IAMMRED>
   ```

4. Use a variable named *$cred* to store the results of using the *Get-Credential* cmdlet. Specify administrator credentials to store in the *$cred* variable. An example of such a command appears here:

   ```
   $cred = Get-Credential iammred\administrator
   ```

5. Use the *Enter-PSSession* cmdlet to open a remote Windows PowerShell console session. Use the credentials stored in the *$cred* variable, and use *localhost* as the name of the remote computer. Such a command appears here:

   ```
   Enter-PSSession -ComputerName localhost -Credential $cred
   ```

6. Notice how the Windows PowerShell console prompt changes to include the name of the remote computer, and also changes the working directory. Such a changed prompt appears here:

   ```
   [localhost]: PS C:\Users\administrator\Documents>
   ```

7. Use the *whoami* command to verify the current context. The results of the command appear here:

```
[localhost]: PS C:\Users\administrator\Documents> whoami

iammred\administrator
```

8. Use the *exit* command to exit the remote session. Use the *whoami* command to verify that the user context has changed.

9. Use WMI to retrieve the BIOS information on the local computer. Use the alternate credentials stored in the *$cred* variable. This command appears here:

```
gwmi -Class win32_bios -cn localhost -Credential $cred
```

   The previous command fails and produces the following error. This error comes from WMI and states that you are not permitted to use alternate credentials for a local WMI connection.

```
gwmi : User credentials cannot be used for local connections
At line:1 char:1
+ gwmi -Class win32_bios -cn localhost -Credential $cred
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [Get-WmiObject], ManagementException
    + FullyQualifiedErrorId : GetWMIManagementException,Microsoft.PowerShell.Commands.
   GetWmiObjectCommand
```

10. Put the WMI command into the *-scriptblock* parameter for *Invoke-Command*. Specify the local computer as the value for *computername* and use the credentials stored in the *$cred* variable. The command appears here (using *-script* as a shortened version of *-scriptblock*):

```
Invoke-Command -cn localhost -script {gwmi -Class win32_bios} -cred $cred
```

11. Press the up arrow key to retrieve the previous command and erase the *credential* parameter. The revised command appears here:

```
Invoke-Command -cn localhost -script {gwmi -Class win32_bios}
```

   When you run the command, it generates the error appearing here because a normal user does not have remote access by default (if you have admin rights, then the command works):

```
[localhost] Connecting to remote server localhost failed with the following error

message : Access is denied. For more information, see the about_Remote_Troubleshooting

Help topic.

    + CategoryInfo          : OpenError: (localhost:String) [], PSRemotingTransport

   Exception

    + FullyQualifiedErrorId : AccessDenied,PSSessionStateBroken
```

12. Create an array of computer names. Store the computer names in a variable named *$cn*. Use the array appearing here:

```
$cn = $env:COMPUTERNAME,"localhost","127.0.0.1"
```

13. Use *Invoke-Command* to run the WMI command against all three computers at once. The command appears here:

```
Invoke-Command -cn $cn -script {gwmi -Class win32_bios}
```

This concludes this step-by-step exercise.

In the following exercise, you will create and receive Windows PowerShell jobs.

Creating and receiving jobs

1. Open the Windows PowerShell console as a non-elevated user.
2. Start a job named *Get-Process* that uses a *-scriptblock* parameter that calls the *Get-Process* cmdlet (*gps* is an alias for *Get-Process*). The command appears here:

```
Start-Job -Name gps -ScriptBlock {gps}
```

3. Examine the output from starting the job. It lists the name, state, and other information about the job. Sample output appears here:

```
Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
9      gps             BackgroundJob   Running       True            localhost
```

4. Use the *Get-Process* cmdlet to determine if the job has completed. The command appears here:

```
Get-Job gps
```

5. Examine the output from the previous command. The *state* reports *completed* when the job has completed. If data is available, the *hasmoredata* property reports *true*. Sample output appears here:

```
Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
9      gps             BackgroundJob   Completed     True            localhost
```

6. Receive the results from the job. To do this, use the *Receive-Job* cmdlet as shown here:

```
Receive-Job gps
```

7. Press the up arrow key to retrieve the *Get-Job* command. Run it. Note that the *hasmoredata* property now reports *false*, as shown here:

```
Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
9      gps             BackgroundJob   Completed     False           localhost
```

8. Create a new job with the same name as the previous job: *gps*. This time, change the *-scriptblock* parameter value to *gsv* (the alias for *Get-Service*). The command appears here:

```
Start-Job -Name gps -ScriptBlock {gsv}
```

9. Now use the *Get-Job* cmdlet to retrieve the job with the name *gps*. Note that the command retrieves both jobs, as shown here:

```
Get-Job -name gps

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
9      gps             BackgroundJob   Completed     False           localhost
11     gps             BackgroundJob   Completed     True            localhost
```

10. Use the *Receive-Job* cmdlet to retrieve the job ID associated with your new job. This time, use the *-keep* switch, as shown here:

```
Receive-Job -Id 11 -keep
```

11. Use the *Get-Job* cmdlet to retrieve your job. Note that the *hasmoredata* property still reports *true* because you're using the *-keep* switch.

This concludes this exercise.

Chapter 4 quick reference

To	Do this
Work interactively on a remote system	Use the *Enter-PSSession* cmdlet to create a remote session.
Configure Windows PowerShell remoting	Use the *Enable-PSRemoting* function.
Run a command on a remote system	Use the *Invoke-Command* cmdlet and specify the command in a *-scriptblock* parameter.
Run a command as a job	Use the *Start-Job* cmdlet to execute the command.
Check on the progress of a job	Use the *Get-Job* cmdlet and specify either the job ID or the job name.
Check on the progress of the newest job	Use the *Get-Job* cmdlet and specify the *-newest* parameter, and supply the number of new jobs to monitor.
Retrieve the results from a job	Use the *Receive-Job* cmdlet and specify the job ID.

Index

Symbols

$$ variable, 142
$acl variable, 362
$args variable, 139, 142, 211, 213
$aryElement variable, 413
$aryLog variable, 554, 556
$aryServer variable, 569
$aryText array, 413
$aryText variable, 413, 416
$aryUsers variable, 566, 567
$ary variable, 151, 154, 158
$bios variable, 354
$caps array, 153
$caption variable, 505
$_ character, 75
$choiceRTN variable, 505
$class variable, 525
$clsID variable, 520
$cn variable, 344, 464
$colDrives variable, 62
$colPrinters variable, 62
$computerName variable, 62, 502, 503
$confirmpreference variable, 216
$constASCII variable, 324
$credential variable, 341, 444, 464
$cred variable, 118, 127
$dc1 variable, 116
$DebugPreference variable, 465
$ (dollar sign) character, 141
$driveData variable, 187, 189
$drives hash table, 527
$dteDiff variable, 329
$dteEnd variable, 329
$dteMaxAge variable, 568
$dteStart variable, 329
$env:psmodulepath variable, 222
$ErrorActionPreference variable, 391, 392, 524, 525, 623
$error.clear() method, 391
$error variable, 142, 191, 389, 390, 392, 624
$ExecutionContext variable, 142
$false variable, 142
$foreach variable, 142
$FormatEnumerationLimit value, 381
$formatEnumeration variable, 225
$help parameter, 184
$HOME variable, 142
$host variable, 97, 142
$input variable, 142, 202, 594
$intGroupType variable, 394, 395
$intSize variable, 568, 570
$intUsers variable, 415
$i++ operator, 415
$i++ syntax, 149
$item variable, 264
$i variable, 143, 148, 152, 328, 390, 415, 547, 566
$LastExitCode variable, 142
$logon variable, 374
$Match variable, 142
$MaximumHistoryCount variable, 594
$message variable, 505
$modulepath variable, 233
$month parameter, 206
$MyInvocation variable, 142
$namespace variable, 524, 525
$newAry variable, 567
$noun variable, 507
$null variable, 142
$num variable, 477, 478, 485, 486, 487, 490
$obj1 variable, 529, 530
$objADSI variable, 384, 413, 415
$objDisk variable, 313
$objEnv variable, 104, 105
$objGroup variable, 395

$objOU variable, 384
$objUser variable, 395, 415
$objWMIServices variable, 322, 328
$objWMI variable, 631
$OFS variable, 142
$oldVerbosePreference variable, 516, 521
$oupath variable, 435
$password variable, 546, 566, 568
$path parameter, 206, 207
$process variable, 138, 264, 345, 364
$profile variable, 268–270, 279
$providername variable, 518, 521
$provider variable, 518
$PSCmdlet variable, 219
$PSHome variable, 142, 267, 272
$PSModulePath variable, 232
$psSession variable, 353
$PSVersionTable variable, 225
$query variable, 326
$rtn variable, 124
$scriptRoot variable, 469, 470
$servers array, 509, 510
$session variable, 345, 352
$share variable, 365
$ShellID variable, 142
$StackTrace variable, 142
$strClass variable, 384, 395, 412, 413, 414, 415
$strComputer variable, 320, 322, 327
$strDatabase variable, 546, 566
$strDomain variable, 410, 546, 566
$strFile variable, 323
$strFname variable, 547, 567
$strLevel variable, 555
$strLname variable, 547
$strLogIdent variable, 555, 556
$strLogPath variable, 569
$strLog variable, 555
$strManager variable, 410
$strName variable, 142, 143, 408, 412, 415
$strOUName variable, 384, 413, 414
$strOU variable, 410, 546, 566, 567
$strPath variable, 142
$strUserName variable, 142
$strUserPath variable, 142
$strUser variable, 410, 415
$this variable, 142
$true variable, 142
$userDomain variable, 62
$userName variable, 62
$users variable, 443
$^ variable, 142
$_ variable, 86, 137, 142, 183, 332
$? variable, 142
$VerbosePreference variable, 210, 516, 519, 521
$verbose variable, 516
$v variable, 381
$wmiClass variable, 320
$wmiFilter variable, 320
$wmiNS variable, 322, 327
$wmiQuery variable, 322, 328
$wshnetwork.EnumPrinterConnections() command, 62
$wshnetwork variable, 61
$xml variable, 563, 565
$year parameter, 206
$zip parameter, 190
[0] syntax, 230
& (ampersand) character, 12
* (asterisk) wildcard operator, 7, 17, 21, 68, 293, 309, 442
' (backtick) character, 137, 480, 628
\ (backward slash), 68
! CALL prefix, 470
^ character, 291
__CLASS property, 188
: (colon), 68
-computername parameter, 108, 118, 124, 246
{ } (curly brackets), missing, 177–178
__DERIVATION property, 188
__DYNASTY property, 188
= (equal) character, 162, 320
= (equal sign), 162, 320
! (exclamation mark), 470
-Force parameter, 459
__GENUS property, 188
' (grave accent) character, 143, 319, 321
> (greater-than) symbol, 320
< (less-than) symbol, 320
__NameSpace class, 287
__NAMESPACE property, 188
`n escape sequence, 328
__PATH property, 188
| (pipe) character, 24, 324, 556
+ (plus symbol), 137, 143
-property argument, 77
__PROPERTY_COUNT property, 188
__provider class, 289
? (question mark), 291
>> (redirect-and-append arrow), 6

> (redirection arrow), 6, 318
__RELPATH property, 188
#requires statement, 234
__SERVER property, 188
! SET keyword, 470
. (shortcut dot), 320
' (single quote) character, 92
__SUPERCLASS property, 188
%windir% variable, 51

A

abstract qualifier, 371
abstract WMI classes, 370
access control lists (ACLs), 90, 362
Access Denied error, 287, 463, 464
Access property, 187
account lockout policy, checking, 430
accounts, user
  creating, 395–396
  deleting, 411–412
AccountsWithNoRequiredPassword.ps1 script, 132
ACLs (access control lists), 90, 362
-action parameter, 488
Active Directory
  cmdlets for
    creating users using, 435–436
    discovering information about forest and domain, 428–431
    finding information about domain controller using, 424–428
  committing changes to, 389
  finding unused user accounts using, 440–442
  installing RSAT for, 420
  locked-out users, unlocking, 436–437
  managing users using, 432–434
  objects in
    ADSI providers and, 385–387
    binding and, 388
    connecting to, 388
    error handling, adding, 392
    errors, 389–392
    LDAP naming convention and, 387–388
    organizational units, creating, 383–384, 413–414
    overview, 383
  objects, updating using Active Directory module, 443–444
  querying, 590
  renaming sites, 431–432
  users
    address information, exposing, 400–401
    computer account, 395–396
    creating, 435–436
    deleting, 411–412
    disabled, finding, 438–439
    finding and unlocking user accounts, 436–437
    general user information, 398–399
    groups, 394–395
    managing, 432–434
    multiple users, creating, 408–409
    multivalued users, creating, 414–417
    organizational settings, modifying, 409–411
    overview, 393–394
    passwords, changing, 444–445
    profile settings, modifying, 403–405
    properties, modifying, 397–398
    telephone settings, modifying, 405–407
    unused user accounts, finding, 440–442
    user account control, 396–397
Active Directory Domain Services. *See* AD DS
Active Directory Management Gateway Service (ADMGS), 419
Active Directory Migration Tool (ADMT), 385
Active Directory module
  automatic loading of, 421
  connecting to server containing, 421–422
  default module locations, 421
  finding FSMO role holders, 422–427
  importing via Windows PowerShell profile, 436
  installing, 419–420
  overview, 419
  updating Active Directory objects using, 443–444
  verifying, 421
Active Directory Service Interfaces (ADSI), 383, 385–387
ActiveX Data Object (ADO), 153
Add cmdlet, 583
Add-Computer cmdlet, 571
Add-Content cmdlet, 84, 571
Add Criteria button, 33
Add-Member cmdlet, 571
AD_Doc.txt file, 431, 462
AddOne filter, 202
AddOne function, 490

Add-Printer cmdlet, 571
Add-PrinterDriver cmdlet, 571
Add-PrinterPort cmdlet, 571
Add-RegistryValue function, 467, 468–469, 470
address information, 400–401
Address tab, Active Directory Users and Computers, 401
AD DS (Active Directory Domain Services)
  AD DS Tool, 385
  deploying
    domain controller, adding to domain, 453–455
    domain controller, adding to new forest, 458–459
    domain controller prerequisites, installing, 457–458
    features, adding, 448
    forests, creating, 452–453
    infrastructure prerequisites, 447
    IP address assignment, 448
    read-only domain controller, adding, 455–457
    renaming computer, 448
    restarting computer, 449
    role-based prerequisites, 448
    script execution policy, setting, 447
    verification steps, 449–450
  tools installation, 448
ADDSDeployment module, 452, 454, 456, 459
AddTwo function, 490
Add-Type cmdlet, 571
Add-WindowsFeature cmdlet, 386, 420, 448, 455, 458
AD LDS Tool, 385
ADMGS (Active Directory Management Gateway Service), 419
admin environment variable, 78, 79
Administrator Audit Logging feature, 557
administrator variable, 100
ADMT (Active Directory Migration Tool), 385
ADO (ActiveX Data Object), 153
ADSI (Active Directory Service Interfaces), 383, 385–387
ADsPath, 384
ADS_UF_ACCOUNTDISABLE flag, 397
ADS_UF_DONT_EXPIRE_PASSWD flag, 397
ADS_UF_DONT_REQUIRE_PREAUTH flag, 397
ADS_UF_ENCRYPTED_TEXT_PASSWORD_ALLOWED flag, 397
ADS_UF_HOMEDIR_REQUIRED flag, 397
ADS_UF_INTERDOMAIN_TRUST_ACCOUNT flag, 397
ADS_UF_LOCKOUT flag, 397
ADS_UF_MNS_LOGON_ACCOUNT flag, 397
ADS_UF_NORMAL_ACCOUNT flag, 397
ADS_UF_NOT_DELEGATED flag, 397
ADS_UF_PASSWD_CANT_CHANGE flag, 397
ADS_UF_PASSWD_NOTREQD flag, 396, 397
ADS_UF_PASSWORD_EXPIRED flag, 397
ADS_UF_SCRIPT flag, 397
ADS_UF_SERVER_TRUST_ACCOUNT flag, 397
ADS_UF_SMARTCARD_REQUIRED flag, 397
ADS_UF_TEMP_DUPLICATE_ACCOUNT flag, 397
ADS_UF_TRUSTED_FOR_DELEGATION flag, 397
ADS_UF_TRUSTED_TO_AUTHENTICATE_FOR_DELEGATION flag, 397
ADS_UF_USE_DES_KEY_ONLY flag, 397
ADS_UF_WORKSTATION_TRUST_ACCOUNT flag, 396, 397
-alias argument, 567
aliases, 489, 626–627
  creating for cmdlets, 19
  finding all for object, 59
  finding for cmdlets, 150–151
  provider for, 66–68
  setting, 246
AllowMaximum property, 315
AllowPasswordReplicationAccountName parameter, 456
AllSigned execution policy, 134
All Users, All Hosts profile, 275–276
AllUsersCurrentHost profile, 269
alphabetical sorting, 77
ampersand (&) character, 12
-a parameter, 212
AppLocker module, 580
Appx module, 580
ArgumentList block, 263
arguments, for cmdlets, 12
[array] alias, 146, 190
Array function, 151
array objects, 54
arrays
  using -contains operator to examine contents of, 507–509
  creating, 589
  indexing, 377
ASCII values, casting to, 152–153
-asjob parameter, 350, 353
-asplaintext argument, 545, 566
assignment operators, 163

association classes, WMI, 370, 373–378
asterisk (*) wildcard operator, 7, 17, 21, 68, 293, 309, 442
ast-write-time property, 30
Attributes property, 82
audit logging (Exchange Server 2010), 557–561
-autosize argument, 313, 327, 331
-AutoSize parameter, 27
Availability property, 187

B

Backspace key, 38
backtick (`) character, 137, 480, 628
backup domain controllers (BDCs), 385
backward slash (\), 68
basename property, 230
BDCs (backup domain controllers), 385
Begin block, 199, 205
BestPractices module, 580
binary SD format, 362
binding, 388
BIOS information, 115, 308–311, 371
bios pattern, 291
BitsTransfer module, 236, 580
BlockSize property, 187
bogus module, 234
[bool] alias, 146, 190
boundary-checking function, 526–527
BranchCache module, 579
breakpoints
  deleting, 494
  enabling and disabling, 494
  ID number, 494
  listing, 492–493
  purpose of, 483
  responding to, 490–492
  script location and, 485
  setting
    on commands, 489–490
    on line number, 483–484
    on variables, 485–489
    overview, 483
  vs. stepping functionality, 483
  storage location, 492
Break statement, 160, 167
business logic
  encapsulating with functions, 194–196
  program logic vs., 194
BusinessLogicDemo.ps1 script, 194
Bypass execution policy, 134
bypass option, 134, 136, 238
[byte] alias, 146, 190

C

canonical aliases, 626–627
Caption property, 187, 315
Case Else expression, 165
casting, 152–153
Catch block, 529. *See also* Try...Catch...Finally blocks
CategoryInfo property, 389
C attribute, 388
-ccontains operator, 507
cd alias, 67
cd .. command, 7
Certificate drive, 102
certificates
  deleting, 74
  finding expired, 75
  listing, 69–73
  provider for, 68
  searching, 74–75
  viewing properties of, 72–73
Certificates Microsoft Management Console (MMC), 69
Certmgr.msc file, 73–74
[char] alias, 146, 190
char data type, 153
chdir alias, 67
Check-AllowedValue function, 526
Checkpoint cmdlet, 584
Checkpoint-Computer cmdlet, 571
Chkdsk method, 187
ChoiceDescription class, 505
choices, limiting. *See* limiting choices
cimclassname property, 380, 381
cimclassqualifiers property, 380
CIM cmdlets
  filtering classes by qualifier, 369–371
  finding WMI class methods, 368–369
  module for, 580
  overview, 367
  retrieving associated WMI classes, 381–382
  using -classname parameter, 367–368
  video classes, 380–381
CIM (Common Information Model), 108, 112, 343–344, 579. *See also* CIM cmdlets

CIM_LogicalDevice class, 362
CIM_UnitaryComputerSystem class, 290
CIMWin32WMI provider, 516
-class argument, 321
Class box, 253
classes
  in WMI, 289–293
  querying WMI, 293–296
  retrieving data from specific instances of, 319–320
  retrieving every property from every instance of, 314
  retrieving specific properties from, 316
-classname parameter, 348, 367–368, 368, 372
-class parameter, 264, 523
__CLASS property, 517
Clear cmdlet, 583
Clear-Content cmdlet, 571
Clear-EventLog cmdlet, 571
Clear-Host cmdlet, 60, 478
Clear-Item cmdlet, 571
Clear-ItemProperty cmdlet, 571
clear method, 392
Clear-Variable cmdlet, 571
ClientLoadableCLSID property, 517
cls command, 21
CLSID property, 517, 519
CMD (command) shell, 1, 76
[cmdletbinding] attribute
  adding -confirm support, 215–216
  adding -whatif support to function, 214–215
  enabling for functions, 210
  for functions, checking parameters automatically, 211–214
  overview, 209, 209–210
  specifying default parameter set, 216–217
  -verbose switch for, 210–211
[CmdletBinding()] attribute, 464, 465
CmdletInfo object, 540
cmdlets. *See also* CIM cmdlets
  Active Directory
    creating users using, 435–436
    finding information about domain controller using, 424–428
    finding locked out users using, 436
    finding unused user accounts using, 440–442
    managing users using, 432–434
  defined, 3
  descriptions of all, 571–578
  displaying graphical command picker of, 52
  execution of
    confirming, 8
    controlling, 7
  finding aliases for, 150–151
  for working with event logs, 587
  most important, 587
  names of, 626–627
  naming, 3, 54–56, 583–586
    verb distribution, 55–56
    verb grouping for, 54–55
  number of on installation, 587
  options for, 12
  overview, 3, 23–24
  searching for using wildcards, 36–39, 43
  suspending execution of, 9
  using Get-Command cmdlet for, 36–39, 43
  verbs for, 174
  with Exchange Server 2010, 539–540
-cmdlets parameter, 559
cn alias, 124, 247
CN attribute, 388
cn parameter, 465
code formatting. *See* formatting code
code, reusing, 178–179
colon (:), using after PS drive name, 68
column heading buttons, 32
-columns argument, 28
command (CMD) shell, 1
commandlets. *See* cmdlets
command-line input, 501
command-line parameter, 502–503
command-line utilities
  exercises using, 20–21
  ipconfig command, 5
  multiple, running, 6
  overview, 4, 5
-command parameter, 489
commands
  most powerful, 588
  setting breakpoints on, 489–490
  whether completed successfully, 592
Commands add-on
  overview, 252–256
  turning off, 256
  using with script pane, 255
command window, prompt for, 76
comments, 179, 627–628
Common Information Model. *See* CIM
-comobject parameter, 50, 61, 62
Compare cmdlet, 584

Compare-Object cmdlet, 571
comparison operators, 162–163
compatibility aliases, 626
Complete cmdlet, 584
Complete-Transaction cmdlet, 571
Compressed property, 187
computer account, 395–396
computer connectivity, identifying, 506
-computername parameter, 182, 293, 344
Concurrency property, 517
ConfigManagerErrorCode property, 187
ConfigManagerUserConfig property, 187
ConfigurationNamingContext property, 431
ConfigureTransportLogging.ps1 script, 557
-Confirm:$false command, 434
-confirm argument, 8–10
Confirm cmdlet, 585
confirmimpact property, 216
ConfirmingExecutionOfCmdlets.txt file, 8
-confirm parameter, 12, 438, 629
-confirm switch, 215–216, 437
Connect cmdlet, 584
connectivity. *See* computer connectivity
Connect-WSMan cmdlet, 571
console, launch options for, 11
ConsoleProfile variable, 280
console window
  copying in, 72
  quotation marks in, 133
constants, 587, 631
  compared with variables, 146
  creating, 170
  creating in scripts, 146
  using, 146–147
-contains operator, 504, 594
  using to examine contents of array, 507–509
  using to test for properties, 509–511
Continue command, 491
Continue statement, 191
Control Properties dialog box, 285
ConversionFunctions.ps1 script, 179
ConversionModuleV6 module, 237
Convert cmdlet, 585
ConvertFrom cmdlet, 584
ConvertFrom-Csv cmdlet, 571
ConvertFromDateTime method, 188
ConvertFrom-Json cmdlet, 571
ConvertFrom-StringData cmdlet, 571
Convert-Path cmdlet, 571
ConvertTo cmdlet, 584
ConvertTo-Csv cmdlet, 572
ConvertToDateTime method, 188
ConvertTo-Html cmdlet, 572
ConvertTo-Json cmdlet, 572
ConvertToMeters.ps1 script, 178
ConvertTo-SecureString cmdlet, 435, 545, 566
ConvertTo-Xml cmdlet, 572
Copy button, Commands add-on, 255
Copy cmdlet, 584
copying from PowerShell window, 72
Copy-Item cmdlet, 230, 279, 572
Copy-ItemProperty cmdlet, 572
Copy-Module function, 229, 231
Copy-Modules.ps1 script, 229, 231, 237, 241, 244
counting backward, 595
-count parameter, 506
count property, 104, 125, 212, 389
CountryCode attribute, 401
country codes, 401–402
CPU (central processing unit), listing processes using
    CPU time criteria, 34
-CreateDnsDelegation parameter, 459
CreateShortCutToPowerShell.vbs script, 141
CreatingFoldersAndFiles.txt file, 80
CreationClassName property, 187
CreationTime property, 82
CreationTimeUtc property, 82
credentials
  -credential parameter, 109, 110, 591
  for remote connection, 339–342
CRSS process, 216
Ctrl+J shortcut, 257
Ctrl+N shortcut, 254, 258
Ctrl+V shortcut, 255, 258
curly brackets ({ }), missing, 177–178
Current Host profile, 268
current property, 202
CurrentUserCurrentHost property, 269, 270
Current User profile, 268
CurrentUser scope, 134

D

-DatabasePath parameter, 459
data types, incorrect, 523–525
date, obtaining current, 75
DateTime object, 205
[DBG] prefix, 495
DC attribute, 388

DDL (dynamic-link library) file, 66
Debug cmdlet, 585
debugging. *See also* errors
  cmdlets for, list of, 483
  functions, 495–496
  scripts, using breakpoints
    deleting breakpoints, 494
    enabling and disabling breakpoints, 494
    exercise, 496–498
    listing breakpoints, 492–493
    responding to breakpoints, 490–492
    setting on commands, 489–490
    setting on line number, 483–484
    setting on variables, 485–489
  using Set-PSDebug cmdlet
    overview, 467
    script-level tracing, 467–471
    stepping through script, 471–479
  strict mode, enabling
    overview, 479
    using Set-PSDebug -Strict, 479–480
    using Set-StrictMode cmdlet, 481–482
-debug parameter, 12, 465
Debug-Process cmdlet, 572
[decimal] alias, 146, 190
DefaultDisplayPropertySet configuration, 294
DEFAULT IMPERSONATION LEVEL key, 307
DefaultMachineName property, 517
DefaultParameterSetName property, 216, 217
default property, 89, 90
default value, setting for registry keys, 95
definition attribute, 86
-definition parameter, 150
Delete method, 412
DeleteUser.ps1 script, 412
deleting
  breakpoints, 494
  users, 411–412
DemoAddOneFilter.ps1 script, 203
DemoAddOneR2Function.ps1 script, 203
DemoBreakFor.ps1 script, 160
DemoDoUntil.vbs script, 154
DemoDoWhile.ps1 script, 151
DemoDoWhile.vbs script, 151
DemoExitFor.ps1 script, 160
DemoExitFor.vbs script, 160
DemoForEachNext.vbs script, 158
DemoForEach.ps1 scrip, 158
DemoForLoop.ps1 script, 156, 157
DemoForLoop.vbs script, 156
DemoForWithoutInitOrRepeat.ps1 script, 156, 157
demoIfElseIfElse.ps1 script, 164
DemoIfElse.vbs script, 163
DemoIf.ps1 script, 161
DemoIf.vbs script, 162
DemoQuitFor.vbs script, 161
DemoSelectCase.vbs script, 164, 166
DemoSwitchArrayBreak.ps1 script, 167
DemoSwitchArray.ps1 scrip, 167
DemoSwitchMultiMatch.ps1 script, 166
DemoTrapSystemException.ps1 script, 191
DemoWhileLessThan.ps1 script, 148, 149
dependencies, checking for modules, 234–236
deploying
  AD DS (Active Directory Domain Services)
    domain controller, adding to domain, 453–455
    domain controller, adding to new forest, 458–459
    domain controller prerequisites, installing, 457–458
    features, adding, 448
    forest, creating, 452–453
    infrastructure prerequisites, 447
    IP address assignment, 448
    read-only domain controller, adding, 455–457
    renaming computer, 448
    restarting computer, 449
    role-based prerequisites, 448
    script execution policy, setting, 447
    verification steps, 449–450
  PowerShell to enterprise systems, 4
deprecated qualifier, 370
__DERIVATION property, 517
-Descending parameter, 35
-description parameter, 187, 260, 315, 627
design considerations, analyzing before development, 94
-detailed argument, 21
DeviceID property, 187
dir alias, 88
DirectAccessClientComponents module, 580
directories
  creating, 82–83
  listing contents of, 81
  listing contents with Get-ChildItem cmdlet, 24–26
    formatting with Format-List cmdlet, 26
    formatting with Format-Table cmdlet, 29
    formatting with Format-Wide cmdlet, 27–29
  properties for, 81–82

DirectoryInfo object, 44
DirectoryListWithArguments.ps1 script, 131–132
DirectoryName property, 82
Directory property, 82
Directory Restore Password prompt, 456
Disable cmdlet, 583
Disable-ComputerRestore cmdlet, 572
Disable-PSBreakpoint cmdlet, 483, 494, 572
Disable-WSManCredSSP cmdlet, 572
Disconnect cmdlet, 584
Disconnect-WSMan cmdlet, 572
-Discover switch, 424
Diskinfo.txt file, 318
disktype property, 146
Dism module, 580
Dismount cmdlet, 585
DisplayCapitalLetters.ps1 script, 153
displaying commands, using Show-Command cmdlet, 52
DisplayName property, 302–303, 432
divide-by-zero error, 492
DivideNum function, 490, 491–492, 492
DnsClient module, 580
DNS Manager tool, 453
DNS server, adding to IP configuration, 453
DNSServerSearchOrder property, 196
Documents and Settings\%username% folder, 141
Do keyword, 154
dollar sign ($), 141, 189
domain controller
  adding to domain, 453–455
  adding to new forest, 458–459
  checking, 430
  prerequisites, installing, 457–458
-DomainMode parameter, 459
-DomainName parameter, 459
DomainNamingMaster role, 425
-DomainNetbiosName parameter, 459
domain password policy, checking, 429
Do statement, 152, 154
dot-sourced functions, using, 182–184
DotSourceScripts.ps1 script, 198
dot-sourcing scripts, 178, 179–181, 180–181
dotted notation, 39, 357
[double] alias, 146, 190
Do...Until statement, 155
DoWhileAlwaysRuns.ps1 script, 155
Do...While statement
  always runs once, 155
  casting and, 152–153
  in VBScript compared with in PowerShell, 151
  range operator, 152
drives
  creating for modules, 232–233
  creating for registry, 87
  for registry, 87–88
  using WMI with, 312–314
DriveType property, 187, 312, 314
dynamic-link library (DLL) file, 66
dynamic qualifier, 370, 371
dynamic WMI classes, 370
__DYNASTY property, 517

E

ea alias, 97, 136
-ea parameter, 27
echo command, 76
-edbFilePath parameter, 551
Else clause, 97, 163, 169, 236
Else If clause, 163
empty parentheses, 105
Enable cmdlet, 583
Enable-ComputerRestore cmdlet, 572
Enabled property, 517
Enable-Mailbox cmdlet, 544, 559
Enable-PSBreakpoint cmdlet, 483, 494, 572
Enable-PSRemoting function, 112
Enable-WSManCredSSP cmdlet, 572
-enddate parameter, 559
EndlessDoUntil.ps1 script, 155
End parameter, 201
Enter cmdlet, 585
Enter in Windows PowerShell option, 71
enterprise systems, deploying PowerShell to, 4
Enter-PSSession cmdlet, 115, 116, 127, 428, 444
EnumNetworkDrives method, 61
EnumPrinterConnections method, 61
Environment PS drive, 77
environment variables
  creating temporary, 78
  deleting, 80
  listing, 77–78
  provider for, 76
  renaming, 79
  viewing using WMI, 330–335
-eq operator, 162
-equals argument, 300, 304
equal sign (=), 162, 320

error[0] variable, 389
-erroraction parameter, 136
-ErrorAction parameter, 12
ErrorCleared property, 187
ErrorDescription property, 187
error handling
  incorrect data types, 523–525
  limiting choices
    using -contains operator to examine contents of array, 507–509
    using -contains operator to test for properties, 509–511
    overview, 504
    using PromptForChoice, 504–505, 534–535
    using Test-Connection to identify computer connectivity, 506
  missing parameters
    assigning value in param statement, 502–503
    detecting missing value and assigning in script, 502
    making parameter mandatory, 503
    overview, 501
  missing rights
    attempt and fail, 512
    checking for rights and exiting gracefully, 513
    overview, 512
  missing WMI providers, 513–523
  out-of-bounds errors
    overview, 526
    placing limits on parameter, 528
    using boundary-checking function, 526–527
  using Try...Catch...Finally
    Catch block, 529
    catching multiple errors, 532–533
    exercise, 536–537
    Finally block, 529–530
error messages
  importance of, 136
  using Trap keyword to avoid confusing messages, 191–192
ErrorMethodology property, 187
ErrorRecord class, 191
ErrorRecord object, 532
errors. *See also* debugging
  command for ignoring, 589
  logic, 466
  run-time, 462–465
  simple typing errors, 479–480
  syntax, 461–462
  terminating vs. nonterminating, 512
-ErrorVariable parameter, 12
escape character (\), 149, 157
-examples argument, 18, 21
Exception property, 532
Exchange Server 2010, 562–565
  audit logging, 557–561
  cmdlets with, 539–540
  logging settings, 553–557
    overview, 553
    transport-logging levels, 554–557
  mailboxes, creating
    multiple mailboxes, 546–547
    using Enable-Mailbox cmdlet, 543–544
    when creating user, 544–546
  message tracking, 568–570
  parsing audit XML file, 562–565
  remote servers, 540–543
  reporting user settings, 548–550
  storage settings
    mailbox database, 550–552
    overview, 550–551
  user accounts, creating
    exercise, 565–568
    when creating mailbox, 544–546
exclamation mark (!), 470
execution policies for scripts
  overview, 134
  required for using profiles, 268
  required for using snippets, 259
  retrieving current, 135–136
  setting, 135–136
execution policy, restricted, 513
execution, unwanted, preventing using While statement, 155–156
Exists property, 82
Exit cmdlet, 585
exit command, 115, 128
Exit For statement, 159
Exit statement, 160–161
ExpandEnvironmentStrings method, 51
expanding strings, 148, 157
expired certificates
  finding, 75
  needed for old executables, 75
explorer filter, 34
Export-Alias cmdlet, 572
Export-CliXML cmdlet, 345, 563, 572
Export cmdlet, 583
Export-Console cmdlet, 11
Export-Csv cmdlet, 572

exportedcommands property, 225
Export-FormatData cmdlet, 572
Export-ModuleMember cmdlet, 241, 248
Export-PSSession cmdlet, 572
Extension property, 82, 193

F

FacsimileTelephoneNumber attribute, 406
FeatureLog.txt file, 450
FileInfo object, 44
-filePath argument, 323
files
  creating, 82–83
  overwriting contents of, 85
  reading from, 84–85
  writing to, 84–85
FileSystemObject, 150
FileSystem property, 187
filesystem provider, 80
FilterHasMessage.ps1 script, 204
Filter keyword, 196, 204
-filter parameter, 199, 312, 326–327, 347, 372, 425, 440, 518, 589
  quotation marks used with, 318
  using to reduce number of returned WMI class instances, 378
filters
  advantages of, 204–205
  overview, 201–203
  performance and, 203–204
  readability of, 204–205
FilterToday.ps1 script, 205
Finally block, of Try...Catch...Finally, 529–530
Find and Replace feature, 622
FindLargeDocs.ps1 script, 196
firewall exceptions, 114
-firstname argument, 568
fl alias, 295
folders
  creating, 82–83
  for user modules, 227–230
  multiple
    creating using scripts, 168–169
    deleting using scripts, 169–170
-force parameter, 81, 82, 94, 112, 134, 269, 279, 434, 440, 545, 552
foreach alias, 143
Foreach alias, 489
ForEach cmdlet, 413, 585
ForEach-Object cmdlet, 137, 159, 183, 287, 292, 381, 382, 489, 550
foreach snippet, 264
Foreach statement
  exiting early, 159–160
  overview, 158
  using from inside PowerShell consule, 159
ForEach statement, 443
-foregroundcolor argument, 328
ForEndlessLoop.ps1 script, 157
-ForestMode parameter, 459
forests
  adding domain controller to, 458–459
  creating, 452–453
For keyword, 156
Format cmdlet, 309, 584
Format-Custom cmdlet, 572
Format-IPOutput function, 200
Format-List cmdlet, 26, 72, 77, 98, 143, 269, 309, 316, 321, 386, 485, 525, 549, 550, 572
Format-NonIPOutput function, 200
*.format.ps1xml files, 371
Format-Table cmdlet, 29, 139, 255, 313, 318, 373, 380, 493, 564, 572
formatting code, 628–629
  constants, 631
  functions, 629–630
  template files, 630
formatting returned data, 189
Format-Wide cmdlet, 572
  alias for, 68
  formatting output with, 27–29
  using, 27–29
For...Next loop, 152
For statement
  flexibility of, 156–157
  in VBScript compared with in PowerShell, 156
  making into infinite loop, 157–158
FreeSpace property, 187, 189
FSMO (Flexible Single Master Operation), 422–427
fsutil utility, 2, 20
ft alias, 295
-full argument, 19, 21
FullName property, 82, 231
FullyQualifiedErrorId property, 389
Function drive, 181
FunctionGetIPDemo.ps1 script, 198
FunctionInfo object, 540
Function keyword, 172, 174, 177, 186, 193, 205, 279

function libraries, creating, 178–179
function notation, 481
function provider, 85
functions
  adding help for
    overview, 184
    using here-string object for, 184–186
  advantages of using, 197–198
  as filters, 201–204
  [cmdletbinding] attribute for, 209–210
    adding -confirm support, 215–216
    adding -whatif support, 214–215
    checking parameters automatically, 211–214
    specifying default parameter set, 216–217
    -verbose switch, 210–211
  comments at end of, 179
  creating, 172
  debugging, 495–496
  delimiting script block on, 177
  dot-sourced, 182–184
  enabling [cmdletbinding] attribute for, 210
  encapsulating business logic with, 194–196
  flexibility of, 198–199
  formatting, 629–630
  including in PowerShell using dot-sourcing, 180–181
  including in scripts, 625
  in VBScript, 171
  listing all, 85–87
  naming, 174–175, 628
  parameters for
    overview, 176
    using more than two, 192–193
    using two input parameters, 186–187
  passing values to, 175
  performance of, 203–204
  readability of, 198
  reusability of, 198
  separating data and presentation activities into different functions, 199–202
  signature of, 195
  type constraints in, 190–191
  using for code reuse, 178–179
  using from imported module, 242–244
  using Get-Help cmdlet with, 243–245
Functions.psm1 module, 239
fw alias, 68

G

gal alias, 45–46
gc alias, 150
gci alias, 79, 85, 333
gcm alias, 37, 238
__GENUS property, 517
-ge operator, 162
Get-Acl cmdlet, 362
Get-ADDefaultDomainPasswordPolicy cmdlet, 429
Get-ADDomain cmdlet, 429
Get-ADDomainController cmdlet, 424, 430
Get-ADForest cmdlet, 428
Get-ADObject cmdlet, 425, 431
Get-ADOrganizationalUnit cmdlet, 435
Get-ADRootDSE cmdlet, 431
Get-ADUser cmdlet, 435, 443, 444
Get-Alias cmdlet, 21, 24, 150, 332, 572
Get-AllowedComputerAndProperty.ps1 script, 511
Get-AllowedComputer function, 508, 509, 510
Get-ChildItem cmdlet, 20, 75, 131, 196, 231, 237, 331, 572
  alias for, 67
  exercises using, 59–60
  listing certificates using, 69
  listing directory contents with, 24–26
  listing registry keys using, 65
Get-Choice function, 505
Get-CimAssociatedInstance cmdlet, 374, 377, 378, 381, 382
Get-CimClass cmdlet, 367–368, 380, 381
Get-CimInstance cmdlet, 183, 246, 343, 353, 371, 373, 381
Get cmdlet, 583
Get-Command cmdlet, 21, 36–39, 43, 56, 172, 238, 242, 421, 423, 579
Get-Command -module <modulename> command, 225
Get-ComputerInfo function, 241, 242
Get-ComputerRestorePoint cmdlet, 572
Get-Content cmdlet, 150, 177, 185, 413, 415, 462–463, 508, 563, 572, 627
Get-ControlPanelItem cmdlet, 572
Get_Count method., 105
Get-Credential cmdlet, 127, 339, 444, 456, 541
Get-Culture cmdlet, 572
Get-Date cmdlet, 20, 329, 572
Get-DirectoryListing function, 192, 193
Get-DirectoryListingToday.ps1 script, 193
Get-Discount function, 194

Get-DiskInformation function, 527
Get-DiskSpace.ps1 script, 189
Get-Doc function, 196
Get-Event cmdlet, 572
Get-EventLog cmdlet, 573, 588
Get-EventLogLevel cmdlet, 553, 555
Get-EventSubscriber cmdlet, 573
Get-ExchangeServer cmdlet, 542
Get-ExCommand cmdlet, 539, 540, 543
Get-ExecutionPolicy cmdlet, 135, 259, 278
Get-FilesByDate function, 194, 205
Get-FilesByDate.ps1 script, 207
Get-FilesByDateV2.ps1 file, 207
GetFolderPath method, 272
Get-FormatData cmdlet, 573
Get-FreeDiskSpace function, 186
Get-FreeDiskSpace.ps1 script, 186
GetHardDiskDetails.ps1 script, 146
Get-Help cmdlet, 58, 68, 243, 245, 540
  creating alias for, 19
  examples using, 21
  overview, 15–20
Get-History cmdlet, 332
Get-Host cmdlet, 573
Get-HotFix cmdlet, 573
GetInfoByZip method, 190
GetIPDemoSingleFunction.ps1 script, 197
Get-IPObjectDefaultEnabledFormatNonIPOutput.ps1 script, 200
Get-IPObjectDefaultEnabled.ps1 script, 199
Get-IPObject function, 199, 200
Get-IseSnippet cmdlet, 261
Get-Item cmdlet, 573
Get-ItemProperty cmdlet, 89, 143, 308, 573
Get-Job cmdlet, 121, 351
Get-Location cmdlet, 68, 573
Get-Mailbox cmdlet, 548
Get-MailboxDatabase cmdlet, 550, 551
Get-MailboxServer cmdlet, 550
Get-MailboxStatistics cmdlet, 558
Get-Member cmdlet, 67, 122, 268, 269, 374, 378, 381, 529, 573
  exercises using, 59–60
  retrieving information about objects using, 44–48
Get-Member object, 376
Get-Module cmdlet, 223, 241
Get-MyBios function, 245, 247, 248
Get-MyBios.ps1 file, 248
Get-MyModule function, 234, 236, 419
Get-MyModule.ps1 script, 236
Get-Net6to4Configuration job, 124
Get-NetAdapter cmdlet, 126, 448, 457
Get-NetConnectionProfile function, 225
Get-OperatingSystemVersion function, 174, 228
Get-OperatingSystemVersion.ps1 script, 174
Get-OptimalSize function, 244
Get-PowerShellRequirements.ps1 script, 3–4
Get-PrintConfiguration cmdlet, 573
Get-Printer cmdlet, 573
Get-PrinterDriver cmdlet, 573
Get-PrinterPort cmdlet, 573
Get-PrinterProperty cmdlet, 573
Get-PrintJob cmdlet, 573
Get-Process cmdlet, 7, 129, 174, 263, 317, 573, 592
Get-Process note* command, 8–9
Get-PSBreakPoint cmdlet, 483, 485, 492, 493, 494, 497, 498, 573
Get-PSCallStack cmdlet, 483, 491, 573
Get-PSDrive cmdlet, 18, 77, 87, 520, 573
Get-PSProvider cmdlet, 66, 67, 573
Get-PSSession cmdlet, 116
Get-Random cmdlet, 573
Get-Service cmdlet, 174, 573
Get-TextStatistics function, 174, 176
Get-TextStats function, 180, 183
Get-TraceSource cmdlet, 573
Get-Transaction cmdlet, 573
Get-TypeData cmdlet, 573
GetType method, 523
Get-UICulture cmdlet, 573
Get-Unique cmdlet, 573
Get-ValidWmiClass function, 523, 524, 525
Get-Variable administrator command, 101
Get-Variable cmdlet, 573
Get-Variable ShellId command, 100
Get-Verb cmdlet, 3, 54, 205, 542
Get-WindowsFeature cmdlet, 385, 386, 420, 448
GetWmiClassesFunction.ps1 script, 184
Get-WmiInformation function, 525
Get-WmiNameSpace function, 286–288
Get-WmiObject cmdlet, 68, 115, 124, 139, 174, 189, 196, 199, 253, 255, 264, 286, 291, 308, 311, 312, 314, 316, 317, 318, 322, 326, 338, 350, 355, 358, 364, 373, 428, 502, 509, 511, 514, 525, 573, 621
Get-WmiProvider function, 289, 516, 521
Get-WSManCredSSP cmdlet, 573
Get-WSManInstance cmdlet, 573
gh alias, 281

G+H keystroke combination, 19
ghy alias, 332, 334
gi alias, 78, 82
globally unique identifier (GUID), 425
gm alias, 122, 292, 361
gmb alias, 248
GPO (Group Policy Object), 4
gps alias, 31, 122, 129
grave accent character (`), 137, 143, 319, 321
greater-than (>) symbol, 320
Group cmdlet, 585
Group-Object cmdlet, 172, 573
group policy, 337–338, 513
Group Policy Object (GPO), 4
groups, 394–395
-groupScope parameter, 433
gsv alias, 32, 130
-gt argument, 59, 61, 162
GUID (globally unique identifier), 425
gwmi alias, 68, 291, 296, 301, 311, 330, 355
gwmi win32_logicaldisk command, 312

H

hard-coded numbers, avoiding, 631
[hashtable] alias, 146, 190
HasMessage filter, 204
hasmoredata property, 129
-Height parameter, 52
Help command, 13–20, 491
Help function, 18, 249
HelpMessage parameter property, 217, 221
here-string object, 184–186
Hit Variable breakpoint, 486
HKEY_CLASSES_ROOT registry hive, 87, 281, 519
HomeDirectory attribute, 404
HomeDrive attribute, 405
HomePhone attribute, 405
HostingModel property, 517
hostname command, 6
HSG key, 93
Hungarian Notation, 631
Hyperv server, 425

I

-icontains operator, 507
IdentifyingPropertiesOfDirectories.txt file, 80
IdentifyServiceAccounts.ps1 script, 323
-identity parameter, 425, 434, 438, 439, 443, 548
-id parameter, 494
IDs for jobs, 120
If statement, 97, 157, 515
  assignment operators, 163
  compared with VBScript's If...Then...End statement, 161
  comparison operators, 162–163
ihy alias, 334
ImpersonationLevel property, 517
Import-Alias cmdlet, 574
Import-Clixml cmdlet, 574
Import cmdlet, 583
Import-Csv cmdlet, 574
importing modules, 241–242
Import-LocalizedData cmdlet, 574
Import-Module cmdlet, 225, 226, 237, 241, 248, 421, 422, 443
Import-PSSession cmdlet, 541, 574
in32_PerfFormattedData_TermService_TerminalServicesSession class, 618
incorrect data types, 523–525
info attribute, 407
InitializationReentrancy property, 517
InitializationTimeoutInterval property, 517
InitializeAsAdminFirst property, 517
Initialize cmdlet, 585
initializing variables, 623
inline code vs. functions, 197–198
InLineGetIPDemo.ps1 script, 196, 197
-inputobject argument, 48, 300, 377, 381
Insert button, 253, 255
Install-ADDSDomainController cmdlet, 454, 456
Install-ADDSForest cmdlet, 459
InstallDate property, 187, 315
-installDNS parameter, 454, 459
installed software, finding, 327–330
installing
  Active Directory module, 419–420
  PowerShell 3.0, 3
  RSAT for Active Directory, 420
InstallNewForest.ps1 script, 452
instance methods, executing
  Invoke-WmiMethod cmdlet, 358–360
  overview, 355–357
  using terminate method directly, 357–358
  [wmi] type accelerator, 360–361
[int] alias, 146, 190
integers, 145

IntelliSense, 256, 462
International module, 580
Internet Protocol (IP) addresses, 112, 196
  adding DNS servers, 453
  assigning, 448
InvocationInfo property, 390
Invoke-AsWorkflow cmdlet, 574
Invoke cmdlet, 583
Invoke-Command cmdlet, 308, 341, 342, 350
  running command on multiple computers using, 118–120
  running single command using, 117–118
Invoke-Expression cmdlet, 574
Invoke-History cmdlet, 281
Invoke-Item cmdlet, 73, 574
Invoke-RestMethod cmdlet, 574
Invoke-WebRequest cmdlet, 68, 574
Invoke-WmiMethod cmdlet, 68, 262, 357, 358–360, 359, 574
Invoke-WSManAction cmdlet, 574
IPAddress property, 196
ipconfig command, 5, 6
IP (Internet Protocol) addresses, 112, 196
  adding DNS servers, 453
  assigning, 448
IPPhone attribute, 406
IPSubNet property, 196
iSCSI module, 580
IscsiTarget module, 580
ise alias, 271
ISE module, 581
ISEProfile variable, 280
IsGlobalCatalog property, 425
isNullOrEmpty method, 443
IsReadOnly property, 82
IsToday filter, 205
i variable, 151
IwbemObjectSet object, 328
iwmi alias, 68
iwr alias, 68

J

jobs
  checking status of, 124–127
  IDs for, 120
  naming, 121–122
  naming return object, 123–124
  overview, 119
  receiving, 120–121, 123–125
  removing, 121
  running, 120
  using cmdlets with, 122–124
Join cmdlet, 584
Join-Path cmdlet, 230, 287, 574
join static method, String class, 593

K

Kds module, 580
-keep parameter, 121, 126, 130, 351
-key parameter, 468
keys, registry
  creating and setting value at once, 95
  creating using full path, 94
  creating with New-Item cmdlet, 93
  listing, 65, 90–91
  overwriting, 94
  setting default value, 95

L

language parser, 461
LastAccessTime property, 82
LastAccessTimeUtc property, 82
LastErrorCode property, 187
LastWriteTime property, 60, 82, 206
LastWriteTimeUtc property, 82
-latest parameter, 176
l attribute, 401
launch options for console, 11
-LDAPFilter parameter, 435
LDAP (Lightweight Directory Access Protocol), 284, 385, 387–388, 425
length property, 30, 150
Length property, 82
-le operator, 162
less-than (<) symbol, 320
Lightweight Directory Access Protocol (LDAP), 284, 385, 387–388, 425
-like operator, 86, 162
Limit cmdlet, 585
Limit-EventLog cmdlet, 574
limiting choices
  using -contains operator to examine contents of array, 507–509
  using -contains operator to test for properties, 509–511

overview, 504
using PromptForChoice, 504–505, 534–535
using Test-Connection to identify computer connectivity, 506
line number, setting breakpoints, 483–484
-list argument, 290
-ListAvailable parameter, 223, 226, 235, 241, 278, 421
List command, 491
listing
certificates, 69–73
directory contents, 81
directory contents with Get-ChildItem cmdlet
formatting with Format-List cmdlet, 26
formatting with Format-Table cmdlet, 29
formatting with Format-Wide cmdlet, 27–29
overview, 24–26
environment variables, 77–78
filtered process list, 34
functions, 85–87
modules, 223–225
providers, 66
registry keys, 65, 90–91
WMI classes, 290–291
ListProcessesSortResults.ps1 script, 132
literal strings, 149
loading modules, 225–227
LocalMachine scope, 134
Local User Management module, 445
locations for modules, 222
-LockedOut parameter, 436
locked-out users, 436–437
logging service accounts, 323–324
logging settings (Exchange Server 2010)
overview, 553
transport-logging levels
configuring, 554–557
reporting, 554–555
logic errors, 466
logon.vbs script, 404
-LogPath parameter, 459
[long] alias, 146, 190
looping
Do...While statement, 152–154
Foreach statement, 159–160
While statement, 150
-lt operator, 162

M

Mailbox2 database, 551
mailboxes (Exchange Server 2010)
creating
using Enable-Mailbox cmdlet, 544
when creating user, 544
database for
examining, 550–551
managing, 551–552
ManagementClass object, 291
mandatory parameter property, 217–218, 503
manifest for modules, 241
-match operator, 59, 162, 291
MaximumAllowed property, 315
MaximumComponentLength property, 187
MD alias, 365
MeasureAddOneFilter.ps1 script, 201
MeasureAddOneR2Function.ps1 script, 204
Measure cmdlet, 584
Measure-Command cmdlet, 574
Measure-Object cmdlet, 313, 574
MediaType property, 187
-Members parameter, 434
MemberType method, 48
-membertype parameter, 46, 47, 81, 122
message tracking (Exchange Server 2010), 568–570
MessageTrackingLogEnabled argument, 569
MessageTrackingLogMaxAge argument, 569
MessageTrackingLogMaxDirectorySize argument, 570
-MessageTrackingLogPath argument, 570
method notation, 481
methods
of WMI classes, 368–369
retrieving for objects using Get-Member cmdlet, 44–48
Microsoft Exchange Server 2010. *See* Exchange Server 2010
Microsoft Management Console (MMC), 69, 386
Microsoft.PowerShell.Diagnostics module, 580
Microsoft.PowerShell.Host module, 581
Microsoft.PowerShell.Management module, 223, 579
Microsoft.PowerShell.Security module, 580
Microsoft.PowerShell.Utility module, 223, 579
Microsoft Systems Center Configuration Manager package, 4
Microsoft TechNet article KB310516, 93
Microsoft TechNet article KB322756, 93

Microsoft TechNet Script Center, 65, 153
Microsoft.WSMan.Management module, 580
missing parameters, handling
  assigning value in param statement, 502–503
  detecting missing value and assigning in script, 502
  making parameter mandatory, 503
  overview, 501
missing rights, handling
  attempt and fail, 512
  checking for rights and exiting gracefully, 513
  overview, 512
missing WMI providers, handling, 513–523
misspelled words, 462, 621
mkdir function, 365
MMAgent module, 580
MMC (Microsoft Management Console), 69, 386
Mobile attribute, 406
-mode parameter, 486, 487
ModifySecondPage.ps1 script, 405
ModifyUserProperties.ps1 script, 398
module coverage, 579–582
-Module parameter, 242, 421
$modulePath variable, 230–231
modules
  checking for dependencies, 234–236
  creating
    manifest for, 241
    overview, 244
    using Get-Help cmdlet with, 243–245
    using Windows PowerShell ISE, 238–239
  creating drive for, 232–233
  deploying providers in, 66
  directory for, 229
  features of, 227
  user folders for, 227–230
  using functions from imported, 242–244
  getting list of, 592
  grouping profile functionality into, 277–278
  importing, 241–242, 244
  installing, 244
  listing all available, 223–225
  listing loaded, 223
  loading, 225–227
  locations for, 222
  $modulePath variable, 230–231
  overview, 222
  using with profiles, 274
  script execution policy required to install, 232
  using from shared location, 237–239
Mount cmdlet, 585
Mount-Database function, 552
Move-ADObject cmdlet, 435
Move cmdlet, 584
Move-Item cmdlet, 574
Move-ItemProperty cmdlet, 574
moveNext method, 202
mred alias, 60
MsDtc module, 579
MSIPROV WMI provider, 516
multiple commands, running, 6
multiple folders
  creating using scripts, 168–169
  deleting using scripts, 169–170
multiple users, creating, 408–409
multivalued users, creating, 414–417
MyDocuments variable, 280
myfile.txt file, 84
Mytestfile.txt file, 20
Mytest folder, 83

N

named parameters, 628
Name input box, 252
-name parameter, 78, 143, 218, 317, 433, 551
Name property, 30, 82, 92, 187, 289, 291, 315, 517
-namespace parameter, 285, 289, 293, 328
__NAMESPACE property, 517
namespaces
  explained, 284
  exploring, 367
  in WMI, 284–288
__namespace WMI class, 517
Name variable, 331
naming
  cmdlets, 3, 54–56, 583–586
    verb distribution, 55–56
    verb grouping for, 54–55
  functions, 174–175, 628
  jobs, 121–122
  return object for job, 123–124
  variables, 631
NDS provider, 385
-ne operator, 162
NetAdapter module, 579
NetBIOS name, 458
NetConnection module, 225, 581
NetLbfo module, 580

NetQos module, 580
NetSecurity module, 579
NetSwitchTeam module, 580
NetTCPIP module, 580
NetworkConnectivityStatus module, 580
network shares, modules from, 237–239
NetworkTransition module, 579
New-ADComputer cmdlet, 432
New-ADGroup cmdlet, 433
New-AdminAuditLogSearch cmdlet, 560, 562
New-ADOrganizationalUnit cmdlet, 432
New-Alias cmdlet, 19, 248, 574
New-CimSession cmdlet, 343
New cmdlet, 583
-newest parameter, 126
New-Event cmdlet, 574
New-EventLog cmdlet, 574
New-ExchangeSession function, 542
New-IseSnippet cmdlet, 259, 260, 630
New-Item cmdlet, 78, 93, 169, 230, 270, 278, 574
New-ItemProperty cmdlet, 574
New-Line function, 180, 183
NewMailboxAndUser.ps1 script, 545
New-Mailbox cmdlet, 539, 545
New-MailBoxDatabase cmdlet, 551, 552
-NewName parameter, 79
New-NetIPAddress cmdlet, 453, 458
New-Object cmdlet, 44, 529, 530, 536, 574
  exercises, 61
  using, 50–51
New-PSDrive cmdlet, 87, 103, 232, 520, 574
New-PSSession cmdlet, 116, 353, 541
New-Service cmdlet, 574
New-TimeSpan cmdlet, 329, 574
New-Variable cmdlet, 100, 168, 324, 574
New-WebServiceProxy cmdlet, 574
New-WSManInstance cmdlet, 575
New-WSManSessionOption cmdlet, 575
Next keyword, 156
NFS module, 579
-noexit parameter, 138, 140
nonterminating errors, 512
-noprofile parameter, 223
notafter property, 75
Notepad.exe file, 7
-notlike operator, 86, 162
-notmatch operator, 162
-not operator, 81, 228, 235
-noun parameter, 42
Novell Directory Services servers, 385
Novell NetWare 3.x servers, 385
NumberOfBlocks property, 188
numbers
  hard-coded, avoiding, 631
  random, generating, 591
NWCOMPAT provider, 385
NwTraders.msft domain, 384, 385, 413

O

O attribute, 388
Object Editor, for Win32_Product WMI class, 518
objects
  finding aliases for, 59
  New-Object cmdlet, 50–51
  retrieving information about using Get-Member cmdlet, 44–48
objFile variable, 147
objFSO variable, 147
objWMIServices variable, 320
-off parameter, 479
ogv alias, 32
On Error Resume Next command, 136
OneStepFurtherWindowsEnvironment.txt file, 335
opening PowerShell, 10, 11
OpenTextFile method, 147
OperationTimeoutInterval property, 517
operators for WMI queries, 321–322
optional modules, 419
-option parameter, 146, 168
options for cmdlets, 12
organizational settings, modifying, 409–411
organizational units (OUs), 4, 383–384, 413, 432
Organization tab, Active Directory Users and Computers, 409, 411
OSinfo.txt file, 319
OtherFacsimileTelephoneNumber attribute, 407
OtherHomePhone attribute, 407
OtherIPPhone attribute, 407
OtherMobile attribute, 407
OtherPager attribute, 407
OtherTelephone attribute, 399
OU attribute, 388
OUs (organizational units), 4, 383–384, 413, 432
-OutBuffer parameter, 12
Out cmdlet, 583
Out-File cmdlet, 324, 575, 592
Out-GridView cmdlet, 31–34, 309, 565, 575
Out-Null cmdlet, 230, 233, 520

out-of-bounds errors, handling
overview, 526
placing limits on parameter, 528
using boundary-checking function, 526–527
Out-Printer cmdlet, 575
output
formatting with Format-Table cmdlet, 29
formatting with Format-Wide cmdlet, 27–29
formatting with Out-GridView cmdlet, 31–34
transcript tool and, 115–116
Out-String cmdlet, 575
-OutVariable parameter, 12

P

Pager attribute, 406
parameter attribute
HelpMessage property, 221
mandatory property, 217–218
overview, 217
ParameterSetName property, 219
position property, 218–219
ValueFromPipeline property, 220–221
parameters
missing, handling
assigning value in param statement, 502–503
detecting missing value and assigning in script, 502
making parameter mandatory, 503
overview, 501
named vs. unnamed, 628
placing limits on, 528
reducing data via, 347–350
ParameterSetName parameter property, 217, 219, 246
Parameters For... parameter box, 254
parameters, function
avoiding use of many, 194
checking automatically, 211–214
using more than two, 192–193
using multiple, 186–187
positional, 96
specifying, 176
specifying default parameter set, 216–217
switched parameters, 193
unhandled, 213–214
param keyword, 465, 502–503
Param statement, 192, 209
Pascal case, 385
-passthru parameter, 137
passwords
changing, 444
domain password policy, checking, 429
Paste button, Command add-on, 255
Paste command, 255
-path parameter, 69, 78, 80, 96, 143, 150, 176, 192, 415, 432, 433
Path property, 315, 359, 517
paths
for module location, 229
for profiles, 267
pause function, 87
PDCs (primary domain controllers), 385
performance, of functions, 203–204
PerLocaleInitialization property, 517
permission issues, 462, 463
PerUserInitialization property, 517
PING commands, 114
PinToStartAndTaskBar.ps1 script, 11
pipe character (|), 24, 75, 324, 556, 622
pipeline, avoiding breaking, 621
PKI module, 580
plus symbol (+), 137, 143
PNPDeviceID property, 188
Pop cmdlet, 585
Pop-Location cmdlet, 93, 96, 575
Popup method, 62
poshlog directory, 448
positional parameters, 96, 175
position message, 136
position parameter property, 218–219
postalCode attribute, 401
postOfficeBox attribute, 401
PowerManagementCapabilities property, 188
PowerManagementSupported property, 188
PowerShell
adding to task bar in Windows 7, 10–11
deploying to enterprise systems, 4
opening, 10, 11
profiles for, 57
PowerShell.exe file, 141
primary domain controllers (PDCs), 385
PrintManagement module, 580
Process block, 200, 203, 205
processes
filtered list of, 34, 35
retrieving list of running processes, 317–318
process ID, 8
Process scope, 134

profileBackup.ps1 file, 279
ProfilePath attribute, 404
profiles
  All Users, All Hosts profile, 275–276
  using central script for, 276–277
  creating, 57, 270–271
  deciding how to use, 271–272
  determining existence of, 270
  grouping functionality into module, 277–278
  using modules with, 274
  using multiple, 273–275
  overview, 267–268
  paths for, 267
  $profile variable, 268–270
  script execution policy required for, 268
  usage patterns for, 272
program logic, 194
ProhibitSendQuota property, 549
PromptForChoice method, 504–505, 534–535
prompt, PowerShell, 76
properties
  using -contains operator to test for, 509–511
  for certificates, 72–73
  for directories, 81–82
  retrieving every property from every instance of class, 314
  retrieving for objects using Get-Member cmdlet, 44–48
  retrieving specific properties from, 316
__PROPERTY_COUNT property, 518
-property parameter, 26, 256, 296, 313, 325, 326, 347, 372, 373, 441
-ProtectedFromAccidentalDeletion parameter, 433
__provider class, 517
ProviderName property, 188
provider property, 90
providers
  alias, 66–68
  certificate, 68
  defined, 65
  environment provider, 76
  filesystem provider, 80
  function provider, 85
  in WMI, 289
  listing, 66
  overview, 65–66
  registry, 90
  variable, 97–98
__provider WMI system class, 517
.ps1 extension, 133
PSComputerName property, 118, 183, 342
Psconsole file, 11
-psconsolefile argument, 12
.psd1 extension, 228
PSDiagnostics module, 580
PSDrives
  for registry, 87–88, 520
  switching, 68
PsisContainer property, 75, 82
.psm1 extension, 228, 237, 239
PSModulePath variable, 229, 421
-PSProvider parameter, 103
PSScheduledJob module, 580
PSStatus property, 188, 295
PSWorkflow module, 581
Pure property, 517
Purpose property, 188
Push cmdlet, 585
Push-Location cmdlet, 93, 575
Put method, 393, 395
pwd alias, 68

Q

-QualifierName parameter, 367, 369
querying
  Active Directory, 590
  WMI
    eliminating WMI query argument, 320–321
    finding installed software, 327–330
    identifying service accounts, 322–323
    logging service accounts, 323–324
    obtaining BIOS information, 308–311
    using operators, 321–322
    overview, 293
    retrieving data from specific instances of class, 319–320
    retrieving default WMI settings, 308
    retrieving every property from every instance of class, 314
    retrieving information about all shares on local machine, 315
    retrieving list of running processes, 317–318
    retrieving specific properties from class, 316
    shortening syntax, 325–326
    specific class, 293–296
    specifying maximum number of connections to server, 316–317
    substituting Where clause with variable, 325

viewing Windows environment variables, 330–335
Win32_Desktop class, 296–298
working with disk drives, 312–314
-query parameter, 314, 348
QuickEdit mode, 72
-quiet parameter, 506
QuotasDisabled property, 188
QuotasIncomplete property, 188
QuotasRebuilding property, 188
quotation marks, 189
in console, 133
used with -filter argument, 318

R

random numbers, 591
range operator, 152
-rate parameter, 195
RDN (relative distinguished name), 384, 387
readability
of filters, 204–205
of functions, 198
Read cmdlet, 585
Read-Host cmdlet, 174, 546, 575, 594
ReadingAndWritingForFiles.txt file, 80
Read mode, 485
read-only variables, 587
ReadUserInfoFromReg.ps1 script
cmdlets used, 143
code, 143–144
variables used, 142
ReadWrite mode, 485
rebooting server, 454, 456
-rebootoncompletion parameter, 459
Receive cmdlet, 584
Receive-Job cmdlet, 120, 123, 129, 350, 353, 354
recipient settings, configuring (Exchange Server 2010)
mailbox, creating
multiple mailboxes, 546–547
using Enable-Mailbox cmdlet, 544
when creating user, 544–546
reporting user settings, 548–550
-recurse parameter, 27, 29, 61, 69, 83, 102, 196, 231
recycled variables, 631
redirect-and-append arrow (>>), 6
redirection arrow (>), 6, 318
red squiggly lines, 462
Regedit.exe file, 90
Register cmdlet, 583
Register-EngineEvent cmdlet, 575
Register-ObjectEvent cmdlet, 575
Register-WmiEvent cmdlet, 575
registry
backing up, 93
determining existence of property, 96
drives for, 87–88
keys for
creating and setting value at once, 95
creating using full path, 94
creating with New-Item cmdlet, 93
overwriting, 94
setting default value, 95
listing keys in, 65, 90–91
modifying property value, 95
modifying property value using full path, 96
provider overview, 90
remote access to, 87
retrieving default property value from, 90
retrieving values from, 89–90
searching for software in, 92
taking care when modifying, 93
testing for property before writing, 97
regular expressions, 591
relative distinguished name (RDN), 384, 387
__RelPath property, 358, 359, 360, 518
RemoteDesktop module, 579
Remote Management firewall exception, 114
remote procedure call (RPC), 338
Remote Server Administration Tools (RSAT), 419
remote servers, 540–543
RemoteSigned execution policy, 134
remoting
accessing local registry, 87
cmdlets for, 107–112
configuring, 112–114
creating session, 115–118
-credential parameter support, 110
firewall exceptions, 114
impersonating current user, 115
running command as different user, 110–111
running single command
on multiple computers, 118–120
on single computer, 117–118
saving sessions, 116–117
testing configuration, 113–114

Windows PowerShell
discovering information about forest and domain, 428–431
obtaining FSMO information using, 428
WMI
disadvantages of, 341
remote results, 344–348
supplying alternate credentials for remote connection, 338–341
using CIM classes to query WMI classes, 343–344
using group policy to configure WMI, 337–338
Remove-ADGroupMember cmdlet, 434
Remove cmdlet, 583
Remove-Computer cmdlet, 575
Remove-Event cmdlet, 575
Remove-EventLog cmdlet, 575
Remove-IseSnippet cmdlet, 261
Remove-Item cmdlet, 74, 80, 83, 169, 279, 575
Remove-ItemProperty cmdlet, 575
Remove-Job cmdlet, 121
Remove-MailboxDatabase cmdlet, 552
Remove-Printer cmdlet, 575
Remove-PrinterDriver cmdlet, 575
Remove-PrinterPort cmdlet, 575
Remove-PrintJob cmdlet, 575
Remove-PSBreakPoint cmdlet, 483, 494, 497, 498, 575
Remove-PSDrive cmdlet, 103, 521, 575
Remove-PSSession cmdlet, 116
Remove-TypeData cmdlet, 575
RemoveUserFromGroup.ps1 script, 434
Remove-Variable cmdlet, 101, 575
Remove-WmiObject cmdlet, 68, 365, 575
Remove-WSManInstance cmdlet, 575
Rename-ADObject cmdlet, 432
Rename cmdlet, 584
Rename-Computer cmdlet, 448, 455, 458, 575
Rename-Item cmdlet, 79, 575
Rename-ItemProperty cmdlet, 575
Rename-Printer cmdlet, 575
renaming environment variables, 79
Repair cmdlet, 585
Repeat command, 491
Replace method, System.String .NET Framework class, 595
-replicationsourcedc parameter, 454
reporting user settings (Exchange Server 2010), 548–550
ReportTransportLogging.ps1 script, 555
requires statement, 246
Reset cmdlet, 585
Reset-ComputerMachinePassword cmdlet, 576
Reset method, 187, 362
Resolve cmdlet, 584
Resolve-Path cmdlet, 576
Resolve-ZipCode function, 190
Resolve-ZipCode.ps1 script, 190
"Resource not available" run-time error, 462
resources, unavailable, 462
Restart cmdlet, 584
Restart-Computer cmdlet, 449, 454, 456, 458, 576
-restart parameter, 448
Restart-PrintJob cmdlet, 576
Restart-Service cmdlet, 576
Restore cmdlet, 585
Restore-Computer cmdlet, 576
Restricted execution policy, 134, 136, 513
resultclassname parameter, 377
Resume cmdlet, 584
Resume-PrintJob cmdlet, 576
Resume-Service cmdlet, 576
RetrieveAndSortServiceState.ps1 script, 139
ReturnValue, 304
returnvalue property, 363
reusability of functions, 198
rich types, 627
rights, missing. *See* missing rights, handling
root/cimv2 WMI namespace, 369, 370
route print command, 6
RPC (remote procedure call), 338
rsat-ad-tools feature, 421
RSAT (Remote Server Administration Tools), 419, 420
Run as different user command, 110–111
Run As Different User dialog box, 111
Run button, 252
Run dialog box, 138
Run ISE As Administrator option, 251
run method, 51
RunningMultipleCommands.txt file, 6
Run Script button, 255
run-time errors, 462–465
rwmi alias, 68

S

sal alias, 67
sAMAccountName attribute, 393, 394
Save cmdlet, 584
sbp alias, 67
sc alias, 67
scheduled tasks, 132
ScheduledTasks module, 580
SchemaMaster role, 425
ScreenSaverExecutable property, 297
ScreenSaverSecure property, 297
ScreenSaverTimeout property, 297
Screen* wildcard pattern, 297
script block, 148
-scriptblock parameter, 128
script execution policies
  overview, 57, 134
  required for using profiles, 268
  required for using snippets, 259
  required to install modules, 232
  retrieving current, 135–136
  setting, 135–136
script-level tracing
  enabling, 467
  trace level 1, 468–469
  trace level 2, 470–471
script pane
  in Windows PowerShell ISE, 254–255
  opening new, 254
  running commands in, 255
  using Commands add-on with, 255
-script parameter, 485, 486, 489
ScriptPath attribute, 404
scripts. *See also* constants; error handling; variables
  advantages of using, 131–133
  using arrays to run commands multiple times, 138
  creating multiple folders using, 168–169
  debugging using breakpoints
    deleting breakpoints, 494
    enabling and disabling breakpoints, 494
    exercise, 496–498
    listing breakpoints, 492–493
    responding to breakpoints, 490–492
    setting on commands, 489–490
    setting on line number, 483–484
    setting on variables, 485–489
  deleting multiple folders using, 169–170
  dot-sourcing, 178, 179–180, 180–181
  enabling support for, 134–135
  execution policies for
    overview, 134, 513
    retrieving current, 135–136
    setting, 135–136
  functions in, 197–198, 625
  using to hold profile information, 276–277
  need for modification of, 196
  overview, 133
  using -passthru parameter, 137–138
  readability of, 627–628
  running, 133
    as scheduled tasks, 132
    inside PowerShell, 140
    outside PowerShell, 140–141
    overview, 138–140
  sharing, 132
  writing, 136–138
SDDL (Security Descriptor Definition Language), 362
SDDLToBinarySD method, 363
SDDLToWin32SD method, 363
Search-ADAccount cmdlet, 436, 437, 438
Search-AdminAuditLog cmdlet, 558
- SearchBase parameter, 440
searching
  certificates, 74–75
  for cmdlets using wildcards, 36–39, 43
secret commands, 132
SecureBoot module, 580
security
  confirming execution of cmdlets, 8
  controlling cmdlet execution, 7
  overview, 6–7
  suspending execution of cmdlets, 9
Security Descriptor Definition Language (SDDL), 362
SecurityDescriptor property, 517
select alias, 293, 296, 340
Select Case statement (VBScript), 164–165
Select cmdlet, 584
Select Columns dialog box, 35
Select-Object cmdlet, 225, 286, 293, 296, 309, 313, 340, 381, 564, 576
Select statement, 316
Select-String cmdlet, 294, 576
Select-Xml cmdlet, 576
Send cmdlet, 584
Send-MailMessage cmdlet, 576
SendTo folder shortcut, 141
-serveraddresses parameter, 453
ServerCore module, 581

ServerManager module, 448, 580
ServerManagerTasks module, 580
-server parameter, 551
__SERVER property, 518
servers, maximum number of connections to, 316–317
service accounts
  identifying, 322–323
  logging, 323–324
ServiceAccounts.txt file, 324
ServiceDependencies.ps1 script, 631
Service Pack (SP) 1, 3
sessions
  creating remote, 115–118
  saving remote, 116–117
Set-ADAccountPassword cmdlet, 435, 444
Set-AdminAuditLog cmdlet, 558
Set-AdminAuditLogConfig cmdlet, 558
Set-ADOObject cmdlet, 432
Set-ADUser cmdlet, 443
set alias, 67
Set-Alias cmdlet, 67, 576
Set cmdlet, 583
Set-Content cmdlet, 67, 576
Set-Date cmdlet, 576
Set-DNSClientServerAddress cmdlet, 453
Set-EventLogLevel cmdlet, 554
Set-ExecutionPolicy cmdlet, 134, 232, 259, 513
SetInfo() method, 389, 393, 396, 414, 416
Set-Item cmdlet, 67, 95, 576
Set-ItemProperty cmdlet, 67, 96, 576
Set-Location cmdlet, 93, 331, 576
  alias for, 67
  switching PS drive using, 68
  working with aliases using, 66
Set-MailboxServer cmdlet, 569
SetPowerState method, 187, 362
Set-PrintConfiguration cmdlet, 576
Set-Printer cmdlet, 576
Set-PrinterProperty cmdlet, 576
Set-Profile function, 279, 280
Set-PropertyItem cmdlet, 95
Set-PSBreakPoint cmdlet, 67, 483, 496, 576
Set-PSDebug cmdlet, 624
  overview, 467
  script-level tracing using
    enabling, 467
    trace level 1, 468–469
    trace level 2, 470–471
  -step parameter, 472–478
  stepping through script, 471–479
  strict mode, enabling, 479–480
Set-Service cmdlet, 576
Set-StrictMode cmdlet, 481–482
Set-StrictMode -Version 2 command, 481
Set-TraceSource cmdlet, 576
Set-Variable cmdlet, 67, 101, 146, 576
Set-WmiInstance cmdlet, 67, 68, 576
Set-WSManInstance cmdlet, 576
Set-WSManQuickConfig cmdlet, 576
shared folders, 237–239
ShareNoQuery.ps1 script, 321
shares, retrieving information about, 315
ShellId variable, 100
shortcut dot (.), 320
shortcuts, adding to SendTo folder, 141
Show cmdlet, 584
Show-Command cmdlet, 52, 576
Show Commands Add-On option, 256
Show-ControlPanelItem cmdlet, 576
Show-EventLog cmdlet, 576
Show MOF button, 361
si alias, 67
signature of functions, 195
SilentlyContinue parameter, 392
simple typing errors, 479–480
[single] alias, 146, 190
single quote (') character, 92, 320
Single-Threaded Apartment model (STA), 273
SIN method, 363
Size property, 188
sl alias, 67, 70, 115, 331
SmallBios.ps1 script, 309
SmbShare module, 580
SmbWitness module, 581
snap-ins
  defined, 66, 222, 234
  uninstalling, 66
snippets
  creating code with, 257–259
  creating user-defined, 259–260
  defined, 257
  removing user-defined, 261–262
  script execution policy required for, 259
software, installed
  finding using WMI, 327–330
  searching for in registry, 92
Software Update Services (SUS), 4
sort alias, 78, 299
Sort cmdlet, 584

sorting
alphabetical listings, 77
list of processes, 35
Sort-Object cmdlet, 139, 298, 302, 322, 576
space, in path of script, 588
sp alias, 67
special variables, 142
spelling, 621
Split cmdlet, 567, 584
split method, 229, 232
Split-Path cmdlet, 576
SP (Service Pack) 1, 3
squiggly lines, 462
Start cmdlet, 583
-startdate parameter, 560
Start-Job cmdlet, 120, 123, 125
Start-Process cmdlet, 577
Start-Service cmdlet, 300, 577
StartService method, 305
Start-Sleep cmdlet, 577
Start Snippets option, 257
Start-Transaction cmdlet, 577
Start-Transcript cmdlet, 58, 115, 273, 591
STA (Single-Threaded Apartment model), 273
state property, 302
static methods, 361–363, 365–366
st attribute, 401
StatusInfo property, 188
status of jobs, checking, 124–127
Status property, 188, 298, 301, 315
Step-Into command, 491
Step-Out command, 491
Step-Over command, 491
-step parameter, 472–478
Stop cmdlet, 491, 584
Stop-Computer cmdlet, 577
Stop-Job cmdlet, 125
StopNotepadSilentlyContinuePassThru.ps1 script, 138
Stop-Process cmdlet, 8–10, 137, 214, 263, 577
Stop-Service cmdlet, 214, 300, 577
Storage module, 579
storage settings (Exchange Server 2010)
mailbox database
examining, 550–551
managing, 551–552
overview, 550–551
streetAddress attribute, 401
Street attribute, 388
strict mode, enabling
overview, 479
using Set-PSDebug -Strict, 479–480
using Set-StrictMode cmdlet, 481–482
-Strict parameter, 480
[string] alias, 146, 190
String Attribute Editor, ADSI Edit, 388
String class, 232
strings
expanding, 148, 157
literal, 149
subject property, 74
subroutines in VBScript, 171
__SUPERCLASS property, 518
supervariable, 79
SupportsDiskQuotas property, 188
SupportsExplicitShutdown property, 517
SupportsExtendedStatus property, 517
SupportsFileBasedCompression property, 188
SupportsQuotas property, 517
SupportsSendStatus property, 517
SupportsShouldProcess attribute, 214, 215
SupportsShutdown property, 517
SupportsThrottling property, 517
suspend argument, 7
Suspend cmdlet, 584
suspending execution of cmdlets, 9
Suspend-PrintJob cmdlet, 577
Suspend-Service cmdlet, 577
SUS (Software Update Services), 4
sv alias, 67
Switch cmdlet, 584
Switch_DebugRemoteWMISession.ps1 script, 465
switched parameters, 193
Switch statement
compared with VBScript's Select Case statement, 164–165
Defining default condition, 165–166
evaluating arrays, 166–167
handling multiple parameters using, 219
matching behavior, controlling, 167
matching with, 166
swmi alias, 67
-syntax argument, 43
syntax errors, 461–462
SystemCreationClassName property, 188
System.Diagnostics.Process .NET Framework object, 122
System.DirectoryServices.DirectoryEntry object, 384
System.Environment .NET Framework class, 272

System.Exception Catch block, 534
System.Exception error, 529, 531
System.IO.DirectoryInfo object, 82
System.IO.FileInfo class, 82, 230
System.Management.Automation.LineBreak .NET Framework class, 483, 485
System.Management.Automation.PSArgumentException object, 532
System.Management.ManagementClass class, 523
System.Math class, 363
SystemName property, 188
SystemSecurity class, 290
System.String class, 229
System.SystemException class, 191
System.Xml.XmlDocument type, 563
-SysVolpath parameter, 459

T

`t command, 588
tab completion, 24, 51, 104, 140
tab expansion, 256, 358, 367, 381, 462–463
TargetObject property, 390
taskbar, adding shortcuts to, 10–11
Tasks menu, 251
TechNet Script Center Script Repository, 445
TechNet Script Repository, 80
TechNet wiki, 257
Tee cmdlet, 584
Tee-Object cmdlet, 577
telephone settings, modifying, 405–407
Telephones tab, Active Directory Users and Computers, 405
template files, 630
terminate method, 355, 357–358, 360
terminating errors, 512
testB object, 391
Test cmdlet, 583
Test-ComputerPath.ps1 script, 506
Test-ComputerSecureChannel cmdlet, 577
Test-Connection cmdlet, 464, 504, 506, 577
Test-Mandatory function, 218
Test-ModulePath function, 228, 231
Test-ParameterSet function, 219
Test-Path cmdlet, 93, 94, 97, 228, 270, 278, 467, 469, 519, 520, 577, 623
Test-PipedValueByPropertyName function, 220
Test-ValueFromRemainingArguments function, 220
Test-WSMan cmdlet, 113, 577
TextFunctions.ps1 script, 180, 183
Text parameter, 260
TextStreamObject, 150
Then keyword, 161
thumbprint attribute, 71
Title parameter, 260
Today parameter, 193
totalSeconds property, 329
Trace cmdlet, 584
Trace-Command cmdlet, 577
-trace parameter, 470
tracing, script-level. *See* script-level tracing
Transcript command, 58
transcript tool, 115–116
transport-logging levels (Exchange Server 2010)
  configuring, 554–557
  reporting, 554–555
Trap statement, 191, 513
triple-arrow prompt, 9
troubleshooting, 621–624
TroubleshootingPack module, 581
TrustedPlatformModule module, 580
Try…Catch…Finally, error handling using
  Catch block, 529
  catching multiple errors, 532–533
  exercise, 536–537
  Finally block, 529–530
  overview, 529
Tshoot.txt file, 6
-type argument, 170
type constraints in functions, 190–191
typename property, 378
Type property, 315
Types.ps1xml file, 294
typing errors, 479–480

U

UAC (User Account Control), 512
UID attribute, 388
unavailable resources, 462
Unblock cmdlet, 584
Unblock-File cmdlet, 577
UNC (Universal Naming Convention), 237, 404, 462
Undefined execution policy, 134
UnderstandingTheRegistryProvider.txt file, 90
UnderstandingTheVariableProvider.txt file, 97
Undo cmdlet, 584
Undo-Transaction cmdlet, 577

unfocused variables, 631
unhandled parameters, 213–214
-unique parameter, 381
Universal Naming Convention (UNC), 237, 404
UnloadTimeout property, 517
Unlock-ADAccount cmdlet, 437, 438
unlocking locked-out users, 436–437
unnamed parameters, 628
Unregister cmdlet, 584
Unregister-Event cmdlet, 577
Unrestricted execution policy, 134
unwanted execution, preventing, 155–156
Update cmdlet, 584
Update-FormatData cmdlet, 577
Update-Help cmdlet, 13–15, 98
UpdateHelpTrackErrors.ps1 script, 14–15
Update-List cmdlet, 577
Update-TypeData cmdlet, 577
UPN (user principal name), 544
url attribute, 399
usage patterns for profiles, 272
UseADCmdletsToCreateOuComputerAndUser.ps1 script, 433
use-case scenario, 501
Use cmdlet, 584
UserAccessLogging module, 580
UserAccountControl attribute, 396
User Account Control (UAC), 512
user accounts, creating (Exchange Server 2010)
  exercise, 565–568
  multiple, 546–547
  when creating mailbox, 544–546
User class, 394
user-defined snippets, 260
UserDomain property, 62
UserGroupTest group, 434
UserNames.txt file, 565
UserName variable, 331
user principal name (UPN), 544
users
  Active Directory and
    computer account, 395–396
    deleting users, 411–412
    exposing address information, 400–401
    general user information, 398–399
    groups, 394–395
    modifying user profile settings, 403–405
    modifying user properties, 397–398
    multiple users, creating, 408–409
    multivalued users, creating, 414–417
    organizational settings, modifying, 409–411
    overview, 393–394
    telephone settings, modifying, 405–407
    user account control, 396–397
  soliciting input from, 594
Use-Transaction cmdlet, 577
UsingWhatif.txt file, 7–8
uspendConfirmationOfCmdlets.txt file, 9

V

ValidateRange parameter attribute, 528
-value argument, 79
ValueFromPipelineByPropertyName property, 217, 220
ValueFromPipeline parameter property, 217, 220–221, 246
ValueFromRemainingArguments property, 217, 220
-value parameter, 324, 468
values
  passing to functions, 175
  retrieving from registry, 89–90
-variable parameter, 485, 486
variables
  constants compared with, 146
  creating, 100–101, 170
  deleting, 101
  grouping, 631
  improperly initialized, 479, 481, 488
  indicating can only contain integers, 145
  initializing properly, 623
  naming, 631
  nonexistent, 479
  provider for, 97–98
  putting property selection into, 373
  recycled, 631
  retrieving, 98–100
  scope of, 631
  setting breakpoints on, 485–489
  special, 142
  storing CIM instance in, 374
  storing remote session as, 116–117
  unfocused, 631
  using, 141–146
  Windows environment variables, 330–335
VariableValue variable, 331
-verb argument, 39
-verbose parameter, 12, 15, 94, 210–211, 227, 516, 519

verbs, 172, 175
  distribution of, 55–56
  grouping of, 54–55
-version parameter, 482
version property, 174, 517
video classes, WMI, 380–381
<view> configuration, 294
VolumeDirty property, 188
VolumeName property, 188
VolumeSerialNumber property, 188
VpnClient module, 580

W

Wait cmdlet, 584
Wait-Event cmdlet, 577
Wait-Job cmdlet, 68, 124, 451
Wait-Process cmdlet, 577
WbemTest (Windows Management Instrumentation Tester), 361, 513
Wdac module, 580
Web Services Description Language (WSDL), 190
Web Services Management (WSMAN), 108
-whatif parameter, 12, 261, 629
  adding support for to function, 214–215
  controlling execution with, 7
  using before altering system state, 74
Whea module, 581
whenCreated property, 441
where alias, 68, 70, 82
Where clause, 325
Where cmdlet, 585
Where-Object cmdlet, 59, 67, 70, 108, 204, 261, 299, 493, 559
  alias for, 68
  compounding, 76
  searching for aliases using, 66
WhileDoesNotRun.ps1 script, 156
While...Not ...Wend loop, 147
WhileReadLine.ps1 script, 150
WhileReadLineWend.vbs script, 147
While statement
  constructing, 148–149
  example of, 150
  looping with, 150
  preventing unwanted execution using, 155–156
While...Wend loop, 147
whoami command, 128
-Width parameter, 52
wildcards
  asterisk (*) character, 7, 17, 21, 68, 293, 309, 442
  in Commands add-on, 252
  in Windows PowerShell 2.0, 226
  loading modules using, 226
  searching for cmdlets using, 36–39
  searching job names, 121
Win32_1394Controller class, 598
Win32_1394ControllerDevice class, 598
Win32_Account class, 614
Win32_AccountSID class, 610
Win32_ACE class, 610
Win32_ActiveRoute class, 607
Win32_AllocatedResource class, 598
Win32_AssociatedBattery class, 601
Win32_AssociatedProcessorMemory class, 598
Win32_AutochkSetting class, 598
Win32_BaseBoard class, 598
Win32_BaseService class, 612
Win32_Battery class, 601
Win32_Bios WMI class, 292, 309, 343, 371, 501, 512, 514, 598
Win32_BootConfiguration class, 608
Win32_Bus class, 598
Win32_CacheMemory class, 598
Win32_CDROMDrive class, 598
Win32_CIMLogicalDeviceCIMDataFile class, 604
Win32_ClassicCOMApplicationClasses class, 603
Win32_ClassicCOMClass class, 603
Win32_ClassicCOMClassSettings class, 603
Win32_ClientApplicationSetting class, 603
Win32_CodecFile class, 607
Win32_CollectionStatistics class, 605
Win32_COMApplication class, 603
Win32_COMApplicationClasses class, 603
Win32_COMApplicationSettings class, 603
Win32_COMClassAutoEmulator class, 603
Win32_COMClass class, 603
Win32_COMClassEmulator class, 603
Win32_ComponentCategory class, 603
Win32_ComputerShutdownEvent class, 607
Win32_ComputerSystem class, 309, 319, 608
Win32_ComputerSystemEvent class, 607
Win32_ComputerSystemProcessor class, 608
Win32_ComputerSystemProduct class, 608
Win32_ComputerSystemWindows ProductActivationSetting class, 615
Win32_COMSetting class, 603
Win32_ConnectionShare class, 612
Win32_ControllerHasHub class, 598

Win32_CurrentProbe class, 601
Win32_CurrentTime WMI class, 294
Win32_DCOMApplicationAccessAllowedSetting class, 603
Win32_DCOMApplication class, 603
Win32_DCOMApplicationLaunchAllowedSetting class, 604
Win32_DCOMApplicationSetting class, 604
Win32_DependentService class, 608
Win32_Desktop class, 296–298, 604
Win32_DesktopMonitor class, 294, 602
Win32_DeviceBus class, 598
Win32_DeviceChangeEvent class, 607
Win32_DeviceMemoryAddress class, 598
Win32_DeviceSettings class, 598
Win32_DFSNode class, 612
Win32_DFSNodeTarget class, 612
Win32_DFSTarget class, 612
Win32_Directory class, 604
Win32_DirectorySpecification class, 604
Win32_DiskDrive class, 598
Win32_DiskDriveToDiskPartition class, 604
Win32_DiskPartition class, 604
Win32_DiskQuota class, 604
Win32_DisplayConfiguration class, 370, 602
Win32_DisplayControllerConfiguration class, 602
Win32_DMAChannel class, 598
Win32_DriverForDevice class, 601
Win32_DriverVXD class, 604
Win32_Environment class, 330, 604
Win32_Fan class, 597
Win32_FloppyController class, 598
Win32_FloppyDrive class, 598
Win32_Group class, 614
Win32_GroupInDomain class, 614
Win32_GroupUser class, 614
Win32_HeatPipe class, 597
Win32_IDEController class, 599
Win32_IDEControllerDevice class, 599
Win32_ImplementedCategory class, 604
Win32_InfraredDevice class, 599
Win32_IP4PersistedRouteTable class, 607
Win32_IP4RouteTable class, 607
Win32_IP4RouteTableEvent class, 607
Win32_IRQResource class, 599
Win32_Keyboard class, 597
Win32_LoadOrderGroup class, 608
Win32_LoadOrderGroupServiceDependencies class, 608
Win32_LoadOrderGroupServiceMembers class, 608
Win32_LocalTime class, 610
WIN32_loggedonuser WMI class, 341
Win32_LogicalDisk class, 146, 187, 189, 318, 605
Win32_LogicalDiskRootDirectory class, 605
Win32_LogicalDiskToPartition class, 605
WIN32_LogicalDisk WMI class, 312, 314
Win32_LogicalFileAccess class, 611
Win32_LogicalFileAuditing class, 611
Win32_LogicalFileGroup class, 611
Win32_LogicalFileOwner class, 611
Win32_LogicalFileSecuritySetting class, 611
Win32_LogicalMemoryConfiguration class, 606
Win32_LogicalProgramGroup class, 612
Win32_LogicalProgramGroupDirectory class, 612
Win32_LogicalProgramGroupItem class, 613
Win32_LogicalProgramGroupItemDataFile class, 613
Win32_LogicalShareAccess class, 611
Win32_LogicalShareAuditing class, 611
Win32_LogicalShareSecuritySetting class, 611
Win32_LogonSession class, 614
Win32_LogonSessionMappedDisk class, 614
Win32_LogonSession WMI class, 374
Win32_LUIDandAttributes class, 605
Win32_LUID class, 605
Win32_MappedLogicalDisk class, 605
Win32_MemoryArray class, 599
Win32_MemoryArrayLocation class, 599
Win32_MemoryDeviceArray class, 599
Win32_MemoryDevice class, 599
Win32_MemoryDeviceLocation class, 599
Win32_ModuleLoadTrace class, 607
Win32_ModuleTrace class, 607
Win32_MotherboardDevice class, 599
Win32_NamedJobObjectActgInfo class, 606
Win32_NamedJobObject class, 605
Win32_NamedJobObjectLimit class, 606
Win32_NamedJobObjectLimitSetting class, 606
Win32_NamedJobObjectProcess class, 606
Win32_NamedJobObjectSecLimit class, 606
Win32_NamedJobObjectSecLimitSetting class, 606
Win32_NamedJobObjectStatistics class, 606
Win32_NetworkAdapter class, 601
Win32_NetworkAdapterConfiguration class, 196, 601
Win32_NetworkAdapterSetting class, 601
Win32_NetworkClient class, 607
Win32_NetworkConnection class, 607
Win32_NetworkLoginProfile class, 614
Win32_NetworkProtocol class, 607
Win32_NTDomain class, 607

Win32_NTEventlogFile class, 614
Win32_NTLogEvent class, 614
Win32_NTLogEventComputer class, 614
Win32_NTLogEventLog class, 614
Win32_NTLogEventUser class, 614
Win32_OnBoardDevice class, 599
Win32_OperatingSystemAutochkSetting class, 605
Win32_OperatingSystem class, 174, 319, 608
Win32_OperatingSystemQFE class, 608
Win32_OSRecoveryConfiguration class, 609
Win32_PageFile class, 606
Win32_PageFileElementSetting class, 606
Win32_PageFileSetting class, 606
Win32_PageFileUsage class, 606
Win32_ParallelPort class, 599
Win32_PCMCIAController class, 599
Win32_PerfFormattedData_ASP_ActiveServerPages class, 615
Win32_PerfFormattedData class, 615
Win32_PerfFormattedData_ContentFilter_IndexingServiceFilter class, 615
Win32_PerfFormattedData_ContentIndex_IndexingService class, 615
Win32_PerfFormattedData_InetInfo_InternetInformationServicesGlobal class, 615
Win32_PerfFormattedData_ISAPISearch_HttpIndexingService class, 615
Win32_PerfFormattedData_MSDTC_DistributedTransactionCoordinator class, 615
Win32_PerfFormattedData_NTFSDRV_SMTPNTFSStoreDriver class, 615
Win32_PerfFormattedData_PerfDisk_LogicalDisk class, 615
Win32_PerfFormattedData_PerfDisk_PhysicalDisk class, 615
Win32_PerfFormattedData_PerfNet_Browser class, 615
Win32_PerfFormattedData_PerfNet_Redirector class, 615
Win32_PerfFormattedData_PerfNet_Server class, 616
Win32_PerfFormattedData_PerfNet_ServerWorkQueues class, 616
Win32_PerfFormattedData_PerfOS_Cache class, 616
Win32_PerfFormattedData_PerfOS_Memory class, 616
Win32_PerfFormattedData_PerfOS_Objects class, 616
Win32_PerfFormattedData_PerfOS_PagingFile class, 616
Win32_PerfFormattedData_PerfOS_Processor class, 616
Win32_PerfFormattedData_PerfOS_System class, 616
Win32_PerfFormattedData_PerfProc_FullImage_Costly class, 616
Win32_PerfFormattedData_PerfProc_Image_Costly class, 616
Win32_PerfFormattedData_PerfProc_JobObject class, 616
Win32_PerfFormattedData_PerfProc_JobObjectDetails class, 616
Win32_PerfFormattedData_PerfProc_ProcessAddressSpace_Costly class, 616
Win32_PerfFormattedData_PerfProc_Process class, 616
Win32_PerfFormattedData_PerfProc_Thread class, 617
Win32_PerfFormattedData_PerfProc_ThreadDetails_Costly class, 617
Win32_PerfFormattedData_PSched_PSchedFlow class, 617
Win32_PerfFormattedData_PSched_PSchedPipe class, 617
Win32_PerfFormattedData_RemoteAccess_RASPort class, 617
Win32_PerfFormattedData_RemoteAccess_RASTotal class, 617
Win32_PerfFormattedData_RSVP_ACSRSVPInterfaces class, 617
Win32_PerfFormattedData_RSVP_ACSRSVPService class, 617
Win32_PerfFormattedData_SMTPSVC_SMTPServer class, 617
Win32_PerfFormattedData_Spooler_PrintQueue class, 617
Win32_PerfFormattedData_TapiSrv_Telephony class, 617
Win32_PerfFormattedData_Tcpip_ICMP class, 617
Win32_PerfFormattedData_Tcpip_IP class, 617
Win32_PerfFormattedData_Tcpip_NBTConnection class, 617
Win32_PerfFormattedData_Tcpip_NetworkInterface class, 617
Win32_PerfFormattedData_Tcpip_TCP class, 617
Win32_PerfFormattedData_Tcpip_UDP class, 618
Win32_PerfFormattedData_TermService_TerminalServices class, 618

Win32_PerfFormattedData_W3SVC_WebService class, 618
Win32_PerfRawData_ASP_ActiveServerPages class, 618
Win32_PerfRawData class, 618
Win32_PerfRawData_ContentFilter_IndexingServiceFilter class, 618
Win32_PerfRawData_ContentIndex_IndexingService class, 618
Win32_PerfRawData_InetInfo_InternetInformationServicesGlobal class, 618
Win32_PerfRawData_ISAPISearch_HttpIndexingService class, 618
Win32_PerfRawData_MSDTC_DistributedTransactionCoordinator class, 618
Win32_PerfRawData_NTFSDRV_SMTPNTFSStoreDriver class, 618
Win32_PerfRawData_PerfDisk_LogicalDisk class, 618
Win32_PerfRawData_PerfDisk_PhysicalDisk class, 618
Win32_PerfRawData_PerfNet_Browser class, 618
Win32_PerfRawData_PerfNet_Redirector class, 618
Win32_PerfRawData_PerfNet_Server class, 619
Win32_PerfRawData_PerfNet_ServerWorkQueues class, 619
Win32_PerfRawData_PerfOS_Cache class, 619
Win32_PerfRawData_PerfOS_Memory class, 619
Win32_PerfRawData_PerfOS_Objects class, 619
Win32_PerfRawData_PerfOS_PagingFile class, 619
Win32_PerfRawData_PerfOS_Processor class, 619
Win32_PerfRawData_PerfOS_System class, 619
Win32_PerfRawData_PerfProc_FullImage_Costly class, 619
Win32_PerfRawData_PerfProc_Image_Costly class, 619
Win32_PerfRawData_PerfProc_JobObject class, 619
Win32_PerfRawData_PerfProc_JobObjectDetails class, 619
Win32_PerfRawData_PerfProc_ProcessAddressSpace_Costly class, 619
Win32_PerfRawData_PerfProc_Process class, 619
Win32_PerfRawData_PerfProc_Thread class, 619
Win32_PerfRawData_PerfProc_ThreadDetails_Costly class, 619
Win32_PerfRawData_PSched_PSchedFlow class, 620
Win32_PerfRawData_PSched_PSchedPipe class, 620
Win32_PerfRawData_RemoteAccess_RASPort class, 620
Win32_PerfRawData_RemoteAccess_RASTotal class, 620
Win32_PerfRawData_RSVP_ACSRSVPInterfaces class, 620
Win32_PerfRawData_RSVP_ACSRSVPService class, 620
Win32_PerfRawData_SMTPSVC_SMTPServer class, 620
Win32_PerfRawData_Spooler_PrintQueue class, 620
Win32_PerfRawData_TapiSrv_Telephony class, 620
Win32_PerfRawData_Tcpip_ICMP class, 620
Win32_PerfRawData_Tcpip_IP class, 620
Win32_PerfRawData_Tcpip_NBTConnection class, 620
Win32_PerfRawData_Tcpip_NetworkInterface class, 620
Win32_PerfRawData_Tcpip_TCP class, 620
Win32_PerfRawData_Tcpip_UDP class, 620
Win32_PerfRawData_TermService_TerminalServices class, 620
Win32_PerfRawData_TermService_TerminalServicesSession class, 620
Win32_PerfRawData_W3SVC_WebService class, 620
Win32_PhysicalMedia class, 598
Win32_PhysicalMemoryArray class, 599
Win32_PhysicalMemory class, 599
Win32_PhysicalMemoryLocation class, 599
Win32_PingStatus class, 506, 607
Win32_PNPAllocatedResource class, 599
Win32_PNPDevice class, 599
Win32_PNPEntity class, 382, 599
Win32_PointingDevice class, 597
Win32_PortableBattery class, 601
Win32_PortConnector class, 599
Win32_PortResource class, 600
Win32_POTSModem class, 602
Win32_POTSModemToSerialPort class, 602
Win32_PowerManagementEvent class, 601
Win32_Printer class, 601
Win32_PrinterConfiguration class, 601
Win32_PrinterController class, 601
Win32_PrinterDriver class, 601
Win32_PrinterDriverDll class, 601
Win32_PrinterSetting class, 602
Win32_PrinterShare class, 612
Win32_PrintJob class, 602
Win32_PrivilegesStatus class, 611
Win32_Process class, 262, 294, 326, 355, 360, 374, 610
Win32_Processor class, 294, 600

Win32_ProcessStartTrace class, 607
Win32_ProcessStartup class, 610
Win32_ProcessStopTrace class, 607
Win32_ProcessTrace class, 607
Win32_Product class, 126, 516, 518
Win32_ProgramGroup class, 613
Win32_ProgramGroupContents class, 613
Win32_ProgramGroupOrItem class, 613
Win32_ProtocolBinding class, 607
Win32_Proxy class, 615
Win32_QuickFixEngineering class, 609
Win32_QuotaSetting class, 605
Win32_Refrigeration class, 597
Win32_Registry class, 610
Win32_ScheduledJob class, 132, 610
Win32_SCSIController class, 600
Win32_SCSIControllerDevice class, 600
Win32_SecurityDescriptor class, 363, 611
Win32_SecurityDescriptorHelper class, 361, 362
Win32_SecuritySettingAccess class, 611
Win32_SecuritySettingAuditing class, 611
Win32_SecuritySetting class, 611
Win32_SecuritySettingGroup class, 611
Win32_SecuritySettingOfLogicalFile class, 611
Win32_SecuritySettingOfLogicalShare class, 611
Win32_SecuritySettingOfObject class, 611
Win32_SecuritySettingOwner class, 611
Win32_SerialPort class, 600
Win32_SerialPortConfiguration class, 600
Win32_SerialPortSetting class, 600
Win32_ServerConnection class, 612
Win32_ServerSession class, 612
Win32_Service class, 294, 301, 373, 612
Win32_SessionConnection class, 612
Win32_SessionProcess class, 612
Win32_ShadowBy class, 613
Win32_ShadowContext class, 613
Win32_ShadowCopy class, 613
Win32_ShadowDiffVolumeSupport class, 613
Win32_ShadowFor class, 613
Win32_ShadowOn class, 613
Win32_ShadowProvider class, 613
Win32_ShadowStorage class, 613
Win32_ShadowVolumeSupport class, 614
Win32_Share class, 315, 612
Win32_ShareToDirectory class, 612
Win32_ShortcutFile class, 605
Win32_SIDandAttributes class, 606
Win32_SID class, 611
Win32_SMBIOSMemory class, 600
Win32_SoundDevice class, 600
Win32_StartupCommand class, 609
Win32_SubDirectory class, 605
Win32_SystemAccount class, 614
Win32_SystemBIOS class, 600
Win32_SystemBootConfiguration class, 609
Win32_SystemConfigurationChangeEvent class, 608
Win32_SystemDesktop class, 609
Win32_SystemDevices class, 609
Win32_SystemDriver class, 604
Win32_SystemDriverPNPEntity class, 600
Win32_SystemEnclosure class, 600
Win32_SystemLoadOrderGroups class, 609
Win32_SystemLogicalMemoryConfiguration class, 606
Win32_SystemMemoryResource class, 600
Win32_SystemNetworkConnections class, 609
Win32_SystemOperatingSystem class, 609
Win32_SystemPartitions class, 605
Win32_SystemProcesses class, 609
Win32_SystemProgramGroups class, 609
Win32_SystemResources class, 609
Win32_SystemServices class, 609
Win32_SystemSetting class, 609
Win32_SystemSlot class, 600
Win32_SystemSystemDriver class, 610
Win32_SystemTimeZone class, 610
Win32_SystemTrace class, 608
Win32_SystemUsers class, 610
Win32_TapeDrive class, 598
Win32_TCPIPPrinterPort class, 602
Win32_TemperatureProbe class, 597
Win32_Thread class, 610
Win32_ThreadStartTrace class, 608
Win32_ThreadStopTrace class, 608
Win32_ThreadTrace class, 608
Win32_TimeZone class, 604
Win32_TokenGroups class, 606
Win32_TokenPrivileges class, 606
Win32_Trustee class, 611
Win32_UninterruptiblePowerSupply class, 601
Win32_USBController class, 600
Win32_USBControllerDevice class, 600
Win32_USBHub class, 600
Win32_UserAccount class, 132, 376, 614
Win32_UserDesktop class, 604
Win32_UserInDomain class, 614
Win32_VideoConfiguration class, 602
Win32_VideoController class, 602

Win32_VideoSettings class, 602
Win32_VoltageProbe class, 601
Win32_VolumeChangeEvent class, 608
Win32_Volume class, 605, 614
Win32_VolumeQuota class, 605
Win32_VolumeQuotaSetting class, 605
Win32_VolumeUserQuota class, 605, 614
Win32_WindowsProductActivation class, 615
windir variable, 77
Windows 7, taskbar shortcuts in, 10–11
Windows 8
  firewall exceptions for, 114
  using -force parameter, 81, 82
  prompts displayed prior to stopping certain processes, 216
  WinRM in PowerShell Client, 112
WindowsDeveloperLicense module, 581
Windows environment variables, 330–335
WindowsErrorReporting module, 581
Windows flag key, 10
Windows Management Framework 3.0 package, 3
Windows Management Instrumentation. *See* WMI
Windows Management Instrumentation Tester (WbemTest), 361
Windows PowerShell. *See* PowerShell
Windows PowerShell 2.0, 226
Windows PowerShell console, 53
Windows PowerShell ISE
  creating modules in, 238–239
  IntelliSense in, 256
  navigating in, 252–254
  running, 251
  running commands in, 255
  script pane in, 254–255
  snippets in
    creating code with, 257–259
    creating user-defined, 259–260
    defined, 257
    removing user-defined, 261–262
  Tab expansion in, 256
Windows PowerShell remoting
  discovering information about forest and domain, 428–431
  obtaining FSMO information using, 428
Windows Remote Management (WinRM), 3
Windows Server 2003, 227
Windows Server 2012, 112
Windows XP, 227
WinNT provider, 385
WinRM (Windows Remote Management), 3
  configuring, 112–114
  firewall exceptions, 114
  overview, 112
  testing configuration, 113–114
wjb alias, 68
WMI classes
  abstract, 370
  association classes, 373–378
  description of, 597–620
  dynamic, 370
  list of, 597–620
  properties of, 597–620
  retrieving WMI instances
    cleaning up output from command, 373
    overview, 371–372
    reducing returned properties and instances, 372–373
  using CIM cmdlets to explore
    filtering classes by qualifier, 369–371
    finding WMI class methods, 368–369
    overview, 367
    retrieving associated WMI classes, 381–382
    using -classname parameter, 367–368
    WMI video classes, 380–381
[wmiclass] type accelerator, 523, 524
WMI cmdlets
  Invoke-WmiMethod cmdlet, 358–360
  overview, 355–357
  using terminate method directly, 357–358
  [wmi] type accelerator, 360–361
WMI Query argument, 320
WMI Tester (WbemTest), 513, 518
[wmi] type accelerator, 189, 360–361
WMI (Windows Management Instrumentation), 1. *See also* WMI classes; WMI cmdlets
  classes in, 289–293
  connecting to, default values for, 307–308
  importance of, 283–284
  missing providers, handling, 513–523
  model for, 284
  namespaces in, 284–288
  obtaining operating system version using, 174
  obtaining specific data from, 189
  providers in, 289
  queries from bogus users, 463
  querying
    eliminating WMI query argument, 320–321
    finding installed software, 327–330

identifying service accounts, 322–323
logging service accounts, 323–324
obtaining BIOS information, 308–311
using operators, 321–322
overview, 293
retrieving data from specific instances of class, 319–320
retrieving default WMI settings, 308
retrieving every property from every instance of class, 314
retrieving information about all shares on local machine, 315
retrieving list of running processes, 317–318
retrieving specific properties from class, 316
shortening syntax, 325–326
specific class, 293–296
specifying maximum number of connections to server, 316–317
substituting Where clause with variable, 325
viewing Windows environment variables, 330–335
Win32_Desktop class, 296–298
working with disk drives, 312–314
remoting
using CIM classes to query WMI classes, 343–344
disadvantages of, 341
using group policy to configure WMI, 337–338
remote results, 344–348
supplying alternate credentials for remote connection, 338–341
using to work with static methods, 361–363, 365–366
WorkingWithVariables.txt file, 97
-Wrap switch, 255
write alias, 68
Write cmdlet, 583
Write-Debug cmdlet, 174, 463, 464, 464–465, 577
Write-Error cmdlet, 174, 577
Write-EventLog cmdlet, 577
Write-Host cmdlet, 178, 328, 488, 577, 592
Write mode, 485
Write-Output cmdlet, 68, 577
Write-Path function, 176
Write-Progress cmdlet, 577, 629
Write-Verbose cmdlet, 209, 519, 520, 577
Write-Warning cmdlet, 577
Wscript.Echo command, 133
Wscript.Quit statement, 161
WSDL (Web Services Description Language), 190
wshNetwork object, 61
wshShell object, 50–52
WS-Management protocol, 112
WSMAN (Web Services Management), 108

X

[xml] alias, 146, 190

About the Author

ED WILSON is a well-known scripting expert who delivers popular scripting workshops to Microsoft customers and employees worldwide. He's written several books on Windows scripting, including *Windows PowerShell™ 2.0 Best Practices*, *Microsoft® Windows PowerShell™ Step By Step*, and *Microsoft® VBScript Step by Step*. Ed is a senior consultant at Microsoft Corporation and writes Hey, Scripting Guy!, one of the most popular TechNet blogs.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press