

# Windows

# POWERSHELL™

4<sup>th</sup> EDITION



TFM®

T.F.M. n.

1. The object referred to in the acronym RTFM, which is a frequent response directed at those who ask questions that are easily answered by reading the documentation.
2. The Foremost Manual
3. A series of books by SAPIEN Press that are the most informative, go-to references for facts and information on a given topic.

Jason Helmick  
Mike F Robbins



# **Windows PowerShell™ TFM®**

*Jason Helmick  
Mike F Robbins*



831 Latour Ct Ste B2  
Napa, CA 94558  
[www.SAPIENPress.com](http://www.SAPIENPress.com)

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and SAPIEN Press was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Windows is a registered trademark, and Windows PowerShell is a trademark, of Microsoft Corporation in the United States and other countries.

The author and publisher have taken care in the publication of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or software programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales.  
For more information, please contact:

SAPIEN Press  
1-866-PRIMALS  
+1 707-252-8700  
[sales@sapien.com](mailto:sales@sapien.com)

Visit SAPIEN Press on the Web: [www.sapienpress.com](http://www.sapienpress.com).

*Library of Congress Cataloging-in-Publication Data*

Helmick, Jason; Robbins, Mike F

Microsoft PowerShell: TFM 4<sup>th</sup> Edition/ Jason Helmick and Mike F Robbins

p. cm.

ISBN 978-0-9821314-6-6 (pbk. : alk. Paper)

1. Microsoft Windows (computer file)      2. Operating systems (Computers)  
3. Windows PowerShell (Computer program language) I. Title

Copyright ©2014 by SAPIEN Technologies, Inc.

All rights reserved. No part of this publication may be reproduced, stored on any retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior express written consent of the publisher.  
Printed in the United States of America.

For information on obtaining permission for use of material from this work, please submit a written request to:

SAPIEN Technologies, Inc.  
Rights and Contracts Department  
831 Latour Ct. Ste. B2  
Napa, CA 94558  
Fax: 707-252-8700

4<sup>th</sup> Edition / 1<sup>st</sup> Printing

*This book is dedicated to everyone whose hard work has made it possible:*

*to Jason and Mike for their writing and research;  
to the gang at HK Communication for their expert editing;  
to Back of the Book for their accurate indexing;  
to David, June, Alex and Ferdinand for their constant and tireless reviewing;  
and to Maricela for her patience and diligence in laying out the book  
and for her creative and beautiful cover art.*

—SAPIEN Press, Publisher



Visit [www.ScriptingAnswers.com](http://www.ScriptingAnswers.com) for answers to all your PowerShell questions!

You can also check out the SAPIEN Technologies blog at <http://blog.sapien.com>,  
where you'll find the latest scripting news, PowerShell tips,  
and much more. Drop by and subscribe today!

Please post any errors that you find to the following web page  
<http://www.sapien.com/errata>

## Contents at a Glance

Before you start . . . . .	01
Diving into PowerShell . . . . .	07
Don't fear the shell . . . . .	23
Finding Help when needed . . . . .	37
Working with providers and drives . . . . .	55
Pipelines for rapid management . . . . .	75
Sorting and measuring data for output . . . . .	103
Filtering your output . . . . .	115
Exporting and formatting your data . . . . .	127
Automation: Your first script . . . . .	153
Increasing your management capabilities . . . . .	167
Storing data for later . . . . .	175
Making your script smarter . . . . .	197
Increasing management with WMI/CIM . . . . .	213
Super-sized management with PowerShell remoting . . . . .	235
Background and scheduled jobs . . . . .	259
Making tools: Turning a script into a reusable tool . . . . .	281
Making tools: Creating object output . . . . .	301
Making tools: Adding Help . . . . .	325
Debugging and error handling . . . . .	341
Creating WinForm tools with PowerShell Studio . . . . .	369
Desired state configuration . . . . .	403
Managing event logs . . . . .	421
Managing files and folders . . . . .	439
Managing permissions . . . . .	465
Managing services . . . . .	479
Managing processes . . . . .	493

Managing the registry .....	507
Regular expressions .....	525
Assignment operators .....	551
Managing Active Directory with ADSI, Part 1 .....	559
Managing Active Directory with ADSI, Part 2 .....	575
Using WMI to manage systems .....	593
Object serialization .....	617
Scope in Windows PowerShell .....	627
Working with the PowerShell host .....	639
Working with XML documents .....	651
Windows PowerShell workflow .....	667



## Foreword

When I learned that SAPIEN was undertaking a massive rewrite and re-organization of *Windows PowerShell: TFM*, I was somewhat nervous. After all, the first three editions by Jeff Hicks and myself were literally the first PowerShell books ever published for their respective versions of the product—I obviously was fond of them! But it was time for an update. Indeed, it was time for a re-do, to a certain extent, because since 2006 we'd learned a lot about how people learned PowerShell and the book needed to reflect newer approaches and the expanding audience. Unfortunately, neither Jeff nor I could find time in our schedules to do the work.

I was delighted when Jason and Mike stepped in. Both have rocketed to the forefront of the PowerShell community in a very short time and they're both great to work with—very knowledgeable and excellent at explaining things in approachable, plain-English terms. Jason has become my right-hand man in running PowerShell.org, Mike was a first-place winner in PowerShell.org's first Scripting Games (after Microsoft handed them off to us)...who better to work on this next generation of TFM?

And I can still say that I had a bit of a hand in things. Jason and Mike kept a lot of the original material, but reworked it into their own style and of course updated it for the newest PowerShell™ versions. Jason's a big fan of my instructional design approach and this is probably the first TFM that really reflects those principles. And certainly, I'll always have a soft spot in my heart for this book and the entire TFM series.

While this might not be the last PowerShell book you ever buy, I know it'll be one that gets heavily dog-eared (or whatever the electronic equivalent is, if you're using an e-book) from constant reference. And, my gratitude to Mike and Jason for upholding the fine tradition that Jeff and I started more than six years ago.

*Don Jones*

When Don and I set out years ago to write the PowerShell TFM books, we had no idea that we were setting a standard for tech books. I was always glad to hear how much the books had helped people out and that they enjoyed, even appreciated, the TFM style and approach. Now, it is time for Jason and Mike to take the baton and lead the pack around the next corner.

If you know, or ever meet Jason and Mike, you might think them an odd pairing. Perhaps no different than Don and I, but it works. As a team, they have your back with this PowerShell thing. Their experiences as IT Pros, trainers, scripters, and community leaders make them ideal candidates to carry the TFM baton. Don and I always wanted the TFM brand to be the books you kept at your desk and this book will be no exception.

Enough of the reminiscing. It is time for you to dig in. Jason and Mike are waiting for you.

*Jeff Hicks*



## About the Authors

**Jason Helmick** is Senior Technologist at Concentrated Tech. His IT career spans more than 25 years of enterprise consulting on a variety of technologies, with a focus on strategic IT business planning. He's a highly successful IT author, columnist, lecturer, and instructor, specializing in automation practices for the IT pro. Jason is a leader in the IT professional community, and serves as board member and COO/CFO of PowerShell.Org and is a Windows PowerShell MVP.

Jason's publications include *Learn Windows IIS in a Month of Lunches*, and he has contributed to numerous industry publications and periodicals, including *PowerShell Deep Dives*, *Microsoft TechNet* and *TechTarget Magazine*. He is a sought-after speaker at numerous technical conferences and symposia, and has been a featured presenter in the Microsoft Virtual Academy. He can be contacted on Twitter: @theJasonHelmick.

**Mike F Robbins** is a Senior Systems Engineer with 20 years of professional experience as an IT Pro. During his career, Mike has provided enterprise computing solutions for educational, financial, health-care, and manufacturing customers. He's a PowerShell MVP and self-proclaimed PowerShell evangelist who uses PowerShell on a daily basis to administer and manage Windows Server, Hyper-V, SQL Server, Active Directory, Exchange, IIS, SharePoint, Terminal Services, EqualLogic Storage Area Networks, AppAssure, and Backup Exec. Mike is the winner of the advanced category in the 2013 PowerShell Scripting Games. He has written guest blog articles for the Hey, Scripting Guy! Blog, PowerShell.org, PowerShell Magazine, and is a contributing author of a chapter in the PowerShell Deep Dives book. Mike is the leader and co-founder of the Mississippi PowerShell User Group. Mike is also a frequent speaker at PowerShell and SQL Saturday technology events. He blogs at mikefrobbins.com and can be found on twitter @mikefrobbins.



# Contents

<b>Contents at a Glance</b>	vi
<b>Foreword</b>	ix
<b>About the Authors</b>	xi
<b>Chapter 1   Before you start</b>	01
Why PowerShell, why now?	01
What Is PowerShell and Why Should I Care?	01
Do I Need to Know All This?	02
Is PowerShell a Good Investment of My Time?	03
What to expect from this book	03
Part 1—The Learning Guide	03
Part 2—The Reference Guide	04
Preparing the perfect lab environment	04
The basic environment	04
The extended/perfect environment	04
The tools you should have on hand	05
<b>Exercise 1 — Before you start</b>	06
<b>Chapter 2   Diving into PowerShell</b>	07
Versions and requirements	07
Installing the Windows Management Framework	08
Launching PowerShell	09
Preparing the shell for you	10
The Options tab	11
The Font tab	11
The Layout tab	11
The Color tab	11
Configuring the ISE	12
Configuring SAPIEN's PrimalScript	13
Installing PrimalScript 2014	13
Getting familiar with PrimalScript	14
Files and Fonts	15
Scripting with PrimalScript	17
Configuring SAPIEN's PowerShell Studio	18
Installing PowerShell Studio	18
Getting familiar with PowerShell Studio	19
Setting the fonts for comfort	20
Scripting with PowerShell Studio	20
<b>Exercise 2 — Preparing your environment</b>	22
<b>Chapter 3   Don't fear the shell</b>	23
Familiar commands	23
The new commands—cmdlets	25

Aliases, the good and the bad .....	26
The downside to aliases .....	27
Command-line assistance .....	29
Command History .....	29
Another history—Get-History .....	30
Line editing .....	31
Copy and paste .....	31
Tab completion .....	31
Keeping a transcript .....	32
<b>Exercise 3 — Don't fear the shell .....</b>	<b>34</b>
<b>Chapter 4   Finding Help when needed .....</b>	<b>37</b>
The extensive and evolving Help system .....	37
Updating Help .....	38
Discovering the commands you need .....	39
Discovering commands .....	40
Parameter switches for more Help .....	41
Another Help system .....	42
Cmdlet anatomy .....	44
How cmdlets work .....	44
Positional parameters .....	46
Shortening parameter names .....	47
Wait a minute! .....	47
Getting into syntax .....	47
Parameter sets .....	47
Syntax .....	48
Stop! How do I know what all these parameters mean? .....	49
SAPIEN Help .....	50
<b>Exercise 4 — Finding Help when needed .....</b>	<b>52</b>
<b>Chapter 5   Working with providers and drives .....</b>	<b>55</b>
PowerShell PSProviders and PSDrives .....	55
PSProviders .....	57
Capabilities .....	58
Getting Help for the provider .....	59
Managing drives and providers .....	60
Mapping drives .....	60
Mapping drives other ways .....	62
Listing child items .....	63
Changing location .....	68
Set-Location .....	69
Cmdlets for working with items .....	70
Copy-Item .....	70
More item cmdlets .....	71
New-Item .....	71
<b>Exercise 5 — Working with providers .....</b>	<b>73</b>
<b>Chapter 6   Pipelines for rapid management .....</b>	<b>75</b>
What does the pipeline do anyway? .....	75

By the way.....	76
Discovering object types, methods, and properties .....	76
Discovering objects .....	77
Selecting just the right information .....	78
Objects after Select-Object .....	80
When you only want a value.....	81
Executing methods on an object.....	81
Caution!.....	83
<b>Exercise 6A — Working with providers .....</b>	<b>86</b>
Understanding how it all works .....	88
What you already know.....	88
How objects are passed .....	89
Parameter binding ByValue.....	90
Parameter binding ByPropertyName .....	92
When ByPropertyName and Properties don't match .....	94
The last resort: parentheticals.....	97
It's all about parentheses .....	98
<b>Exercise 6B — Working with providers .....</b>	<b>101</b>
<b>Chapter 7   Sorting and measuring data for output .....</b>	<b>103</b>
Sorting your results .....	103
Displaying only unique information .....	106
Case Sensitivity .....	107
Select and sort a subset of data .....	108
Measuring the results .....	110
<b>Exercise 7 — Sorting and measuring data .....</b>	<b>113</b>
<b>Chapter 8   Filtering your output .....</b>	<b>115</b>
You should filter your data .....	115
Comparison operators.....	117
Text comparison only .....	119
Logical operators.....	121
More on Where-Object .....	122
Best performance for filtering.....	123
<b>Exercise 8 — Filtering your output.....</b>	<b>125</b>
<b>Chapter 9   Exporting and formatting your data.....</b>	<b>127</b>
Useful reporting and more!.....	127
Exporting and importing comma-separated value (CSV) files.....	127
Opening the CSV in your favorite application .....	130
More about CSV files .....	131
Import- and Export- other delimited data .....	132
Exporting and importing XML .....	132
Export-CliXML .....	132
Comparing objects and collections .....	134
Text1.txt .....	134
Text2.txt .....	135

Real-world comparison example .....	135
Converting to CSV and XML .....	136
Converting objects to HTML .....	138
Dressing it up .....	139
Formatting object for reports .....	141
Formatting rules overview: When does PowerShell use a list or a table? .....	141
Formatting lists and tables .....	143
Format-List .....	143
How did we know? .....	144
Format-Table .....	144
Format-Wide .....	146
Grouping information .....	147
By the way .....	148
The grid view .....	148
<b>Exercise 9 — Exporting and formatting your data .....</b>	<b>152</b>
<b>Chapter 10   Automation: Your first script .....</b>	<b>153</b>
Getting ready for your first script .....	153
Script files .....	153
Scripting security features .....	154
Why won't my scripts run? .....	154
When scripts don't run .....	154
What's an execution policy? .....	156
Execution policies .....	156
Setting an execution policy .....	156
Digitally signing scripts .....	157
Trusted scripts .....	159
Digitally signing scripts .....	159
Is PowerShell dangerous? .....	160
Safer scripts from the Internet .....	160
Using PrimalScript/PowerShell Studio to sign scripts .....	162
Signing with PrimalScript .....	162
Signing scripts with PowerShell Studio .....	162
Turning your one-liners into scripts .....	163
The process .....	165
<b>Exercise 10 — Automating your first script .....</b>	<b>166</b>
<b>Chapter 11   Increasing your management capabilities .....</b>	<b>167</b>
Extending PowerShell's management capabilities .....	167
It's a matter of time .....	168
Snap-ins and modules .....	169
Adding snap-ins .....	169
Importing modules .....	170
Sometimes you need to manually import modules .....	171
<b>Exercise 11 — Extending PowerShell .....</b>	<b>173</b>
<b>Chapter 12   Storing data for later .....</b>	<b>175</b>
Learning the ropes .....	175
Defining variables in PowerShell .....	175

What about scripting? .....	178
Creating variables with cmdlets .....	179
Get-Variable .....	180
Set-Variable/New-Variable .....	181
Clear-Variable .....	182
Remove-Variable .....	183
Forcing a data type .....	183
Variable with multiple objects (arrays) .....	187
Array or collection .....	187
Working with arrays .....	189
Associative arrays .....	190
Creating an associative array .....	191
Using an associative array .....	192
Handling variables inside of quotes .....	192
Subexpressions .....	193
What's the point? .....	194
<b>Exercise 12 — Storing data.</b> .....	<b>196</b>
<b>Chapter 13   Making your script smarter</b> .....	<b>197</b>
Making decisions (If) .....	197
Making decisions better (switch) .....	202
Using the Foreach loop .....	204
Using the For loop .....	206
Using the While loop .....	207
Using the Do/While loop .....	208
Using the Do/Until loop .....	208
Break and Continue .....	210
<b>Exercise 13 — Making your script smarter.</b> .....	<b>212</b>
<b>Chapter 14   Increasing management with WMI/CIM.</b> .....	<b>213</b>
What is WMI? .....	213
WMI structure .....	213
Get-WMIOBJECT versus Get-CIMInstance .....	214
Navigating and using WMI/CIM .....	215
Navigating with the WMI Explorer .....	216
Working with classes .....	217
Working with WMI/CIM objects and properties .....	220
Executing WMI/CIM methods .....	222
Increasing your reach to remote computers .....	225
Security and end-points with WMI .....	226
<b>Exercise 14 — Increasing management with WMI.</b> .....	<b>233</b>
<b>Chapter 15   Super-sized management with PowerShell remoting</b> .....	<b>235</b>
Why you need this now .....	235
The Windows Management Framework .....	235
How remoting works—the WS-MAN protocol .....	236
Enabling PowerShell remoting .....	236
Managing one-to-one .....	238
Managing one-to-many .....	239

Remote command output .....	241
Establishing sessions .....	242
Disconnected sessions .....	244
Getting cmdlets anytime you need them .....	246
Advanced session configurations .....	248
Handling multi-hop remoting .....	252
<b>Exercise 15 — PowerShell remoting .....</b>	<b>257</b>
<b>Chapter 16   Background and scheduled jobs .....</b>	<b>259</b>
What's a job? .....	259
Local background jobs .....	260
Remote background jobs .....	261
Managing jobs .....	263
Storing output from jobs .....	268
Scheduled jobs .....	270
Options for scheduled jobs .....	271
Triggers for scheduled jobs .....	274
Registering and using scheduled jobs .....	275
<b>Exercise 16 — PowerShell jobs .....</b>	<b>279</b>
<b>Chapter 17   Making tools: Turning a script into a reusable tool .....</b>	<b>281</b>
Making a better script with parameters .....	281
Turning a script into a function .....	283
Variables and scope .....	287
Advanced functions: Turning a function into an advanced function .....	287
Adding parameter attributes .....	289
Mandatory .....	289
ValueFromPipeline .....	290
ValidateNotNullOrEmpty .....	291
ValidateRange .....	292
DontShow .....	293
Scripts versus modules .....	294
Turning your functions into a module .....	294
Adding a manifest for your module .....	295
<b>Exercise 17 — Create a reusable tool .....</b>	<b>299</b>
<b>Chapter 18   Making tools: Creating object output .....</b>	<b>301</b>
Taking your cmdlets to the next level .....	301
Output designed for the pipeline .....	301
Creating a custom object .....	305
Properties and methods .....	311
Creating custom views .....	313
<b>Exercise 18 — Creating custom output .....</b>	<b>323</b>
<b>Chapter 19   Making tools: Adding Help .....</b>	<b>325</b>
Why should I add Help? .....	325
The Help documentation for the operator .....	325
Comment-based Help .....	325

Comment-based Help for functions .....	327
Comment-based Help for scripts .....	332
Parameter Help.....	333
Inline documentation with Write- cmdlets.....	335
Write-Verbose .....	335
Write-Debug.....	337
<b>Exercise 19 — Adding comment-based Help .....</b>	<b>340</b>
<b>Chapter 20   Debugging and error handling .....</b>	<b>341</b>
The purpose of a quality debugger.....	341
What could go wrong? .....	341
PowerShell debugger.....	342
Setting BreakPoints.....	343
About \$ErrorActionPreference.....	345
Generic error handling .....	346
Using -ErrorAction and -ErrorVariable .....	346
Inline debugging with Write-Debug and Write-Verbose .....	350
Handling error output.....	355
Using Try-Catch-Finally .....	355
Displaying errors with Write-Warning and Write-Error .....	360
Storing errors in a log for later .....	362
<b>Exercise 20 — Debugging and error handling.....</b>	<b>368</b>
<b>Chapter 21   Creating WinForm tools with PowerShell Studio .....</b>	<b>369</b>
What are Windows Forms? .....	369
Making WinForms easy with SAPIEN PowerShell Studio .....	369
Using SAPIEN PowerShell Studio—A practical example .....	370
Common controls .....	371
Events and controls.....	379
<b>Exercise 21 — Create a WinForm tool .....</b>	<b>402</b>
<b>Chapter 22   Desired state configuration .....</b>	<b>403</b>
What is desired state configuration? .....	403
Configuration .....	404
Resources .....	405
Configuration reusability.....	409
DSC configuration MOF file .....	409
Deployment .....	412
<b>Exercise 22 — Desired state configuration.....</b>	<b>419</b>
<b>InDepth 1   Managing event logs .....</b>	<b>421</b>
Windows 7 and Later .....	427
Working with remote event logs.....	430
Configuring event logs .....	432
Backup event logs .....	432
Location, location, location.....	433
Clearing event logs .....	434

<b>InDepth 2   Managing files and folders . . . . .</b>	<b>439</b>
Creating text files . . . . .	439
Reading text files . . . . .	441
Parsing text files . . . . .	443
Parsing IIS log files . . . . .	443
Parsing INI files . . . . .	446
Copying files . . . . .	448
Provider alert . . . . .	450
Deleting files . . . . .	450
Renaming files . . . . .	451
File attributes and properties . . . . .	452
Setting attributes . . . . .	454
It isn't necessarily what you think . . . . .	455
Proceed with caution . . . . .	456
Working with paths . . . . .	457
Test-Path . . . . .	457
Convert-Path . . . . .	457
Split-Path . . . . .	458
Resolve-Path . . . . .	460
Creating directories . . . . .	462
Listing directories . . . . .	463
Deleting directories . . . . .	463
<b>InDepth 3   Managing permissions . . . . .</b>	<b>465</b>
Viewing permissions . . . . .	465
Viewing permissions for an entire object hierarchy . . . . .	469
Changing permissions . . . . .	470
Automating Cacls.exe to change permissions . . . . .	472
Complex permissions in PowerShell . . . . .	473
Get-Owner . . . . .	473
SetOwner . . . . .	475
Retrieving access control . . . . .	476
Removing a rule . . . . .	478
<b>InDepth 4   Managing services . . . . .</b>	<b>479</b>
Listing services . . . . .	479
Elevated sessions . . . . .	482
Starting services . . . . .	484
Stopping services . . . . .	485
Suspending and resuming services . . . . .	485
Restarting services . . . . .	486
Managing services . . . . .	487
Change service logon account . . . . .	490
Controlling services on remote computers . . . . .	491
Change service logon account password . . . . .	492
<b>InDepth 5   Managing processes . . . . .</b>	<b>493</b>
Detailed process information . . . . .	495
Starting a process . . . . .	497
Stopping local processes . . . . .	497
Waiting on a process . . . . .	498
Process tasks . . . . .	498

Find process details . . . . .	499
Find process owners . . . . .	500
Remote processes . . . . .	502
Creating a remote process . . . . .	505
Stopping a remote process . . . . .	506
<b>InDepth 6   Managing the registry . . . . .</b>	<b>507</b>
Creating registry items . . . . .	512
Removing registry items . . . . .	514
Standard registry rules apply . . . . .	514
Searching the registry . . . . .	515
Managing remote registries with WMI . . . . .	517
Enumerating keys . . . . .	518
Enumerating values . . . . .	518
Modifying the registry . . . . .	522
Create a string value: . . . . .	522
Create a DWORD: . . . . .	522
Create an expanded string value: . . . . .	522
Create a multistring value: . . . . .	522
Managing remote registries with the .NET Framework . . . . .	523
<b>InDepth 7   Regular expressions . . . . .</b>	<b>525</b>
Writing regular expressions . . . . .	527
UNC . . . . .	531
IP addresses . . . . .	533
Named character sets . . . . .	534
Select-String . . . . .	538
Regex object . . . . .	539
Replace operator . . . . .	542
Regular-expression examples . . . . .	546
Email address . . . . .	546
There's more than one way . . . . .	546
String with no spaces . . . . .	548
Telephone number . . . . .	549
<b>InDepth 8   Assignment operators . . . . .</b>	<b>551</b>
<b>InDepth 9   Managing Active Directory with ADSI, Part 1 . . . . .</b>	<b>559</b>
Using Active Directory with ADSI . . . . .	559
ADSI fundamentals . . . . .	559
ADSI queries . . . . .	560
Using ADSI objects . . . . .	561
Retrieving ADSI objects . . . . .	561
Searching for ADSI objects . . . . .	567
Working with ADSI objects . . . . .	573
<b>InDepth 10   Managing Active Directory with ADSI, Part 2 . . . . .</b>	<b>575</b>
Managing Active Directory with ADSI . . . . .	575
Working with users by using the [ADSI] type adapter . . . . .	576
Obtaining password age . . . . .	580

Deleting users . . . . .	581
Bulk-creating users . . . . .	582
Working with computer accounts . . . . .	584
Delete computer accounts . . . . .	585
Working with groups . . . . .	586
Moving objects . . . . .	587
Searching for users . . . . .	587
Fun with LDAP Filters . . . . .	588
Microsoft Active Directory cmdlets . . . . .	590
<b>InDepth 11   Using WMI to manage systems . . . . .</b>	<b>593</b>
Why all these cmdlets . . . . .	593
Retrieving basic information . . . . .	594
Listing available classes . . . . .	596
Listing properties of a class . . . . .	597
Examining existing values . . . . .	598
Getting information using SAPIEN's WMI Explorer . . . . .	599
Remote management . . . . .	600
Change Service Logon Account . . . . .	601
The [WMI] type . . . . .	603
The [WMISearcher] type . . . . .	605
Invoking WMI methods . . . . .	606
Practical examples . . . . .	607
WMI events and PowerShell . . . . .	610
Register-WmiEvent . . . . .	610
Get-Event . . . . .	611
Remove-Event . . . . .	613
Unregister-Event . . . . .	613
Querying for specific events . . . . .	613
Watching services . . . . .	614
Watching files and folders . . . . .	615
Read more about it . . . . .	616
<b>InDepth 12   Object serialization . . . . .</b>	<b>617</b>
Why export objects to XML? . . . . .	619
Jobs . . . . .	621
Remote management . . . . .	621
Workflow . . . . .	624
<b>InDepth 13   Scope in Windows PowerShell . . . . .</b>	<b>627</b>
Types of scope . . . . .	627
Scope-aware elements . . . . .	628
Scope rules . . . . .	628
Specifying scope . . . . .	630
By the way . . . . .	630
Best practices for scope . . . . .	631
Forcing best practices . . . . .	633
Dot sourcing . . . . .	634
By the way . . . . .	636
Nested prompts . . . . .	636

Tracing complicated nested scopes .....	637
Special scope modifiers .....	638
<b>InDepth 14   Working with the PowerShell host .....</b>	<b>639</b>
Culture clash .....	641
Using the UI and RawUI .....	642
Reading lines and keys .....	642
Changing the window title .....	644
Changing colors .....	645
Changing window size and buffer .....	646
Nested prompts .....	647
By the way .....	648
Quitting PowerShell .....	649
Prompting the user to make a choice .....	649
<b>InDepth 15   Working with XML documents .....</b>	<b>651</b>
What PowerShell does with XML .....	651
Basic XML manipulation .....	653
A practical example .....	657
Turning objects into XML .....	663
<b>InDepth 16   Windows PowerShell workflow .....</b>	<b>667</b>
<b>Index .....</b>	<b>683</b>



## Chapter 1

# Before you start

### Why PowerShell, why now?

When PowerShell released, the only publicly known software to embed its management with PowerShell was the upcoming release of Microsoft Exchange Server 2007. Today, many platforms are integrating their management through the use of PowerShell—and the list continues to grow.

Here are a few reasons why you should learn to use PowerShell:

1. It's not going away.
2. Most Microsoft products use it.
3. You can't do everything from the GUI.
4. It can make your life easier through automation.
5. Microsoft certification exams contain PowerShell questions.
6. If you don't learn it, someone else will.

### What Is PowerShell and Why Should I Care?

Administrators of UNIX and Linux systems (collectively referred to as “\*nix” throughout this book) have always had the luxury of administrative scripting and automation. In fact, most \*nix operating systems are built on a command-line interface (CLI). The graphical operating environment of \*nix systems—often the “X Windows” environment—is itself a type of shell; the operating system is fully functional without this graphical interface. This presents a powerful combination. Since the operating

system is typically built from the command-line, there isn't anything you can't do from an administrative sense, from the command-line. That's why \*nix admins are so fond of scripting languages like Python and Perl—they can accomplish real administration tasks with them.

Windows, however, has always been different. When a Microsoft product group sat down to develop a new feature—say, the Windows DNS Server software—they had certain tasks that were simply required. First and foremost, of course, was the actual product functionality—such as the DNS Server service, the bit of the software that actually performs as a DNS server. Some form of management interface was also required and the Windows Common Engineering Criteria specified that the minimum management interface was a Microsoft Management Console (MMC) snap-in—that is, a graphical administrative interface. If they had extra time, the product team might create a Windows Management Instrumentation (WMI) provider, connecting their product to WMI, or they might develop a few command-line utilities or Component Object Model (COM) objects, allowing for some scriptable administrative capability. Rarely did the WMI or COM interfaces fully duplicate all the functionality available in the graphical console. This often meant that some administrative tasks could be accomplished via the command-line or a language like VBScript, but you couldn't do everything that way. You'd always be back in the graphical console for something, at some point.

Not that graphical interfaces are bad, mind you. After all, they're how Microsoft has made billions from the Windows operating system. Yet clicking buttons and selecting check boxes can only go so fast, and with commonly performed tasks like creating new users, manual button-clicking is not only tedious, it's prone to mistakes and inconsistencies. Administrators of \*nix systems have spent the better part of a decade laughing at Windows' pitiable administrative automation and third parties have done very well creating tools like AutoIt or KiXtart to help fill in the gaps for Windows' automation capabilities.

That's no longer the case, though. PowerShell is now a part of the Microsoft Common Engineering Criteria and it occupies a similar position of importance with product groups outside the Windows operating system. Now, administrative functionality is built in PowerShell, first. Any other form of administration, including graphical consoles, utilizes the PowerShell based functionality. Essentially, graphical consoles are merely script wizards that run PowerShell commands in the background to accomplish whatever they're doing. This is becoming more evident with the release of Windows 8 and Server 2012. Exchange Server 2007 was the first example of this—the graphical console simply runs PowerShell commands to do whatever corresponds to the buttons you click (the console even helpfully displays the commands it's running, so you can use those as examples to learn from).

## Do I Need to Know All This?

Yes. Look, clearly graphical user interfaces are easier to use than command-line utilities. PowerShell's consistent naming and architecture make it easier to use. But in the end, it's all about the command-line. A Windows admin who operates from the command-line can create a hundred new users in the time it takes to create just one in the graphical user interface. That's an efficiency savings managers just can't ignore. PowerShell lets you perform tasks, en masse, that can't be done at all with the GUI, like updating the password a particular service uses to log in to dozens of computers.

If you've been with Windows since the NT 4.0 days, you may remember a time when earning your Microsoft Certified Systems Engineer (MCSE) certification was not only a way to differentiate yourself from the rest of the admins out there, it was also a ticket to a \$20,000 or more pay raise. Those days, of course, are gone. Today, management is looking at production applicable skills to differentiate the highly paid admins from the entry level ones. PowerShell is the skill management is after. Try finding an IT manager who'll pay top dollar for a \*nix admin who can't script in Perl or Python or some similar language. Before long, Windows managers will have figured it out, too—a PowerShell savvy admin can do more work, in less time, and with fewer mistakes, than an administrator who doesn't know PowerShell. That's the type of bottom line, dollars and cents criteria that any smart manager can understand. So, yes, you need to know this.

## Is PowerShell a Good Investment of My Time?

Absolutely. That can be tough to believe, given all the scripting technologies that Microsoft has inflicted on the world in the past and then quickly abandoned—KiXtart, VBScript, batch files, JScript, and the list goes on. But PowerShell is different. First of all, PowerShell is currently in version 4.0, but there will be a version 5.0 and a version 6.0... So while today's PowerShell isn't perfect, it's pretty darn good, and in almost all ways it's already better than any similar technology we've had in the past.

Yet, as I have already described, PowerShell is here to stay. Microsoft can't walk away from PowerShell as easily as they did VBScript, primarily because so many products today are being built on top of PowerShell. With PowerShell embedded in Exchange, SharePoint, the System Center family, Windows Server 2008 R2, and Windows Server 2012, it's a safe bet that you're going to be working with PowerShell for a decade or more, at least. In computer time, that's about a century, so it's definitely a good investment.

## What to expect from this book

We wanted this book to be more than just another PowerShell reference guide. In the process of updating this book to the new version of PowerShell, some changes to the overall format were made. We think you'll like them.

The first change is obvious in that this book has been updated to the new version of PowerShell. Don't worry, the same great content from the previous versions of the book is still included and has been updated.

The second and largest change—the book is now separated into two parts: a Learning Guide and a Reference Guide.

### Part 1—The Learning Guide

The Learning Guide was created from years of experience teaching and using PowerShell. PowerShell has a steep learning curve and it is an important aspect of this update to give you a path to quickly learn PowerShell. PowerShell has many features and capabilities; many of these you need to know well to be successful in solving day to day problems and perform automation tasks.

When this book was first written, many readers already had experience with other scripting languages, like VBScript. Today, admins with little or no experience are finding that PowerShell can help them and that they need a way to learn from the ground up. The Learning Guide was designed with the admin in mind, taking you through the basics—into the deep corners—and bringing you out on top of the most powerful management tool available for Microsoft environments.

Each chapter includes exercises, to give you a chance to try out your new skills. These exercises were designed with real world applications in mind. There isn't a lot of help because as you will find out, PowerShell can provide you with everything you need to know.

We also want you to know that the exercises in this book have been tested and used for years by the authors. They are cryptic—like real life. The exercises are not step by step. They are designed to start you thinking about *how* to solve the problem and *how* to discover the solution. This may seem frustrating at first, but try it out. If you work through them, you will build the necessary skills with PowerShell. The process of thinking through the problem will be worth it!

The exercises are short, so take some extra time to experiment on your own. There is nothing better than trying to apply these concepts in your own environment.

## Part 2—The Reference Guide

The reference portion of the guide still contains many of the same great topics from the previous versions of the book. The original information is still very relevant today and we didn't want to lose it. Additional topics have been added to give you updates about the latest version of PowerShell.

Try to finish the Learning Guide, first, before you dive into the Reference Guide. Many of the topics are advanced and assume that you already are comfortable with the information in the Learning Guide section. The Reference Guide also contains additions such as workflow, which was first introduced in PowerShell version 3. Many of these additions won't be necessary for you to know when you first get started. In fact, you may be solving problems and creating automation without this information entirely. The idea is that if there comes a time when you need this information, it will be waiting for you.

## Preparing the perfect lab environment

The best way to learn PowerShell is by using PowerShell. This is the primary reason we added exercises. Learning a tool like PowerShell takes practice and experimentation. I'm sure you don't want to experiment on your production environment, so a lab environment not connected to production is best. We have tried to make the requirements simple, but you can extend your environment to gain more experience and test some of your own ideas.

Let's start with the basic environment.

### The basic environment

To start learning PowerShell and using the exercises in this book requires nothing more than PowerShell on your laptop or computer. You should be using the latest version, which can be downloaded from Microsoft. You are looking for the Windows Management Framework 4—I know the title doesn't have PowerShell in its name, but more on that later. If you don't already have PowerShell installed, we will give you some notes and download locations in the next chapter.

Better than using your local computer is to create a Virtual Machine (VM), with Windows Server 2012 R2 or Windows 8.1. This way you can experiment without any fear of hurting your local computer or deleting files. We always keep a VM around for quick testing and experimentation. You can always download the evaluation software for Windows Server 2012 R2 or Windows 8.1 from [www.microsoft.com](http://www.microsoft.com).

### The extended/perfect environment

The extended environment requires a little more setup. You can learn PowerShell by using the basic environment, so don't feel like you have to take a lot of time and build this extended environment before you get started. While you're learning PowerShell, keep this alternative environment in mind, as it will give you a greater platform to experience PowerShell before you try it out in a production environment.

This environment consists of 2-4 virtual machines (VMs) to recreate a small production environment. Here is what we recommend:

- 1 Domain controller running on Windows Server 2012 R2.
2. Windows 8.1 client with the remote administration tools installed, joined to the domain.
3. Optionally, additional servers and products, such as Microsoft Exchange 2013. Make the environment similar to the one you want to manage.

You can download the trial versions of the operating systems from Microsoft, so you don't have to worry about licenses. Install the domain controller and Windows 8.1 using Microsoft Hyper-V or VMWare, whichever is your preferred virtualization platform.

## The tools you should have on hand

If you have a PowerShell console, then you are ready to get started—but keep in mind that you will be learning to automate and script, so you'll need some scripting tools.

With PowerShell installed, you will quickly find the PowerShell Integrated Scripting Environment (ISE). It's a free scripting tool from Microsoft and we will show you around the ISE later in the book. We will also be showing you a variety of additional tools from SAPIEN Technologies—professional scripting tools designed to help you.

---

### Note from the Authors:

We have used SAPIEN scripting tools for years and we wouldn't start a project without them.

We are not trying to sell you these tools—we just want you to see how they can help. There are no free tools that can help you accomplish some of your goals—it requires a group of people dedicated to understanding and developing a suite of tools for admins and developers alike. SAPIEN has done that. OK, we'll stop now. ;)

Most of the additional tools seen in this book can be downloaded from [www.SAPIEN.com](http://www.SAPIEN.com)

While you can install the tools as you go along, I recommend you download and install the trial versions of the following:

- SAPIEN PowerShell Studio 2014
- SAPIEN PrimalScript 2014
- SAPIEN WMI Explorer 2014

Let's try our first exercise, it will be simple, to make sure you're ready to get started learning PowerShell!

## Exercise 1 — Before you start

This is an easy exercise and it gives you a chance to get everything together before you dive into the rest of the book. Take a few minutes and complete the following tasks.

### Task 1

Decide what lab environment you would like to use. You can change your mind later, but we recommend at least a single VM that does not have network access to your production environment.

Do not worry about launching PowerShell yet or if you have the correct version—we will address that in the next chapter, but your best operating system option is Windows Server 2012 R2 or Windows 8.1

### Task 2

If a VM is impractical for you and you just want to start using PowerShell on your laptop, take precaution to do no harm to your production environment.

### Task 3

You can download the SAPIEN evaluation versions for PrimalScript and PowerShell Studio, plus the rest of the SAPIEN tools. Don't worry about installing them yet, as you will do that in the next chapter.

When you're ready to get started, move on to the next chapter and start diving into PowerShell.

## Chapter 2

# Diving into PowerShell

### Versions and requirements

PowerShell version 4 is the current version and it is included in Windows 8.1 and Server 2012 R2. This doesn't mean that you have to upgrade your operating system (OS) to get the new version of PowerShell—you can download and install it on previous client and server operating systems. PowerShell version 4 is supported on the following operating systems:

- Windows 8.1
- Windows Server 2012 R2
- Windows 7 with SP1
- Windows Server 2008 with SP2
- Windows Server 2008 R2 with SP1

How do you know which version of PowerShell is currently installed on your computer? Open a PowerShell console and type the following:

```
$PSVersionTable
```

```
Administrator: Windows PowerShell
PS C:\> $PSVersionTable
Name          Value
----          -----
PSVersion      4.0
WSManStackVersion 3.0
SerializationVersion 1.1.0.1
CLRVersion    4.0.30319.34003
BuildVersion   6.3.9600.16394
PSCompatibleVersions {1.0, 2.0, 3.0, 4.0}
PSRemotingProtocolVersion 2.2

PS C:\> -
```

Figure 2-1

There is a lot of useful information provided by **\$PSVersionTable** and as you progress through the book, it will become more relevant. For now, check the value listed for **PSVersion**. If it doesn't show any output or if you received an error, you might be running PowerShell version 1. Regardless, if you don't have the latest version of PowerShell, it's time to fix that.

## Installing the Windows Management Framework

PowerShell is installed as a hotfix with other technologies, such as the Windows Remote Management service, that you will need when using PowerShell. To find a copy of PowerShell for your computer, open a web browser, visit <http://download.microsoft.com>, and then search for “Windows Management Framework 4.0.” This is the package that includes PowerShell version 4.

---

### Note:

If you have an operating system that does not support PowerShell version 4, then download version 3. If you’re unlucky enough to still be running Windows XP, you could download version 2, but you should upgrade your OS instead.

Before you start downloading a package, check the system requirements section for the latest notes regarding PowerShell on your OS. If you are going to update the version of PowerShell on your servers, notice the warnings in the system requirements. At the time of this writing, certain enterprise products don’t work well with PowerShell version 4 and you would not want to make this mistake on a production server. So, check the system requirements again just to make sure.

**Note:**

All operating systems require the Microsoft .NET Framework 4.5 or later to support the features of PowerShell version 4.

Once you verify and update any missing requirements, you're ready to download the Windows Management Framework package. You will notice 32-bit and 64-bit packages during the download, labeled as **x86** and **x64**. Make sure you choose the right version for your machine.

After the installation is complete, PowerShell will be located in the C:\Windows\System32\WindowsPowerShell\V1.0 directory. The V1.0 is the version of the language engine, not the version of PowerShell. This is where you will find the files PowerShell.exe and PowerShell\_ISE.exe.

## Launching PowerShell

There are several ways to launch PowerShell, depending on your operating system. For example, the PowerShell icon is pinned to the Taskbar on the graphical versions of Windows Server 2008 R2 and Windows Server 2012 R2. With Windows 7, you have to dig a little deeper and select **Start/All Programs/Accessories/WindowsPowerShell**, where you will see the program icons. If you're using Windows 8.1, start typing "PowerShell" on the Start screen and the program will appear in your search.

In all cases, we recommend that you pin PowerShell to your Taskbar for fast access, but which one? If you are on a 64-bit operating system, you will notice that you have two versions of PowerShell, one labeled **x86** and one labeled **x64**. In almost all cases, you will want to launch the 64-bit version. If you are on a 32-bit operating system, then the only PowerShell icon you will see is the **x86**.

If your computer has User Account Control (UAC) enabled (and it should), PowerShell will launch without administrative credentials. This will be a problem if you need to perform administrative tasks, so right-click the PowerShell icon and launch using **Run As Administrator**.

---

**Note:**

Make sure to launch PowerShell with administrative credentials to perform administrative tasks. Right-click the PowerShell icon and select Run As Administrator.

You can launch PowerShell from the command line by typing PowerShell.exe. In fact, there are several switch options available that will be discussed throughout this book. You can see the switch options by typing the following:

```
PowerShell.exe /?
```

If you're working on the non-graphical versions of the Windows Server products, known as Core, there are no icons, Taskbars, or Start menus. Simply type PowerShell.exe into the command window and PowerShell will launch.

## Preparing the shell for you

There are two components to PowerShell, the text based console and the visual Integrated Scripting Environment (ISE). We will look at the ISE in a moment, but you will want to customize the console application. You will spend a lot of time staring at the text based console, and the default font and size are not comfortable for most people.

Start by opening the properties of the console window by clicking the control box (the PowerShell icon in the upper left of the window), and then selecting **Properties**. You will notice four tabs similar to those shown in Figure 2-2.



Figure 2-2

## The Options tab

First, make sure that **QuickEdit Mode** is selected under **Edit Options**. Many of the enhanced PowerShell consoles on products such as Microsoft Exchange and System Center do not have this option selected by default.

## The Font tab

PowerShell uses almost all of the characters on your keyboard for syntax. A font type and size that's too small will make it hard to tell the difference between characters such as a backtick (`), a single quote('), open parenthesis, and curly brace ({ }), to name a few. In fact, you might be staring at this book right now trying to see them! The key is to adjust the font type and size to something that is comfortable to read and large enough to identify those characters.

We wish we could just tell you the best answer, but in this case it's totally dependent on your monitor, resolution settings, and personal preference. The default font, Raster Fonts, is difficult for most people to read. A better choice is a TrueType font, such as Lucida Console or Consolas.

The next selection is the font size. The larger the font, the easier it is to identify small characters on the screen. Notice that as you increase the font size, the **Windows Preview** displays how your console will look. If the console window exceeds the size of the box, you will need to correct that with settings on the **Layout Tab**.

## The Layout tab

When PowerShell displays output, it will fill the screen buffer size with data. This includes both width and height. If your buffer size is larger than the actual window size, you will not see the output—you might even believe that there was no data to display.

First, set the width for both the **Window Size** and **Screen Buffer Size** to the same value. You want to make sure that the window does not extend beyond your monitor and hide results from you. You can use the **Windows Preview** to check your work.

Next, set the **Screen Buffer Size** for the height to 3000 or greater. Some PowerShell management consoles for products such as Microsoft Exchange leave this at the old default of 300, which is not enough to buffer the output. You will see a vertical scroll bar, so you can scroll up and down through your results.

## The Color tab

PowerShell has a default color scheme of a blue background with white text. The **Color** tab allows you to customize your color scheme. To make the printing easier for this book, we change the default to a white background with black text.

## Configuring the ISE

The ISE is a graphical host for PowerShell that provides an improved console experience, with syntax coloring and IntelliSense, and a basic script writing editor. While this editor does not replace a professional scripting environment, it is certainly better than using Notepad.

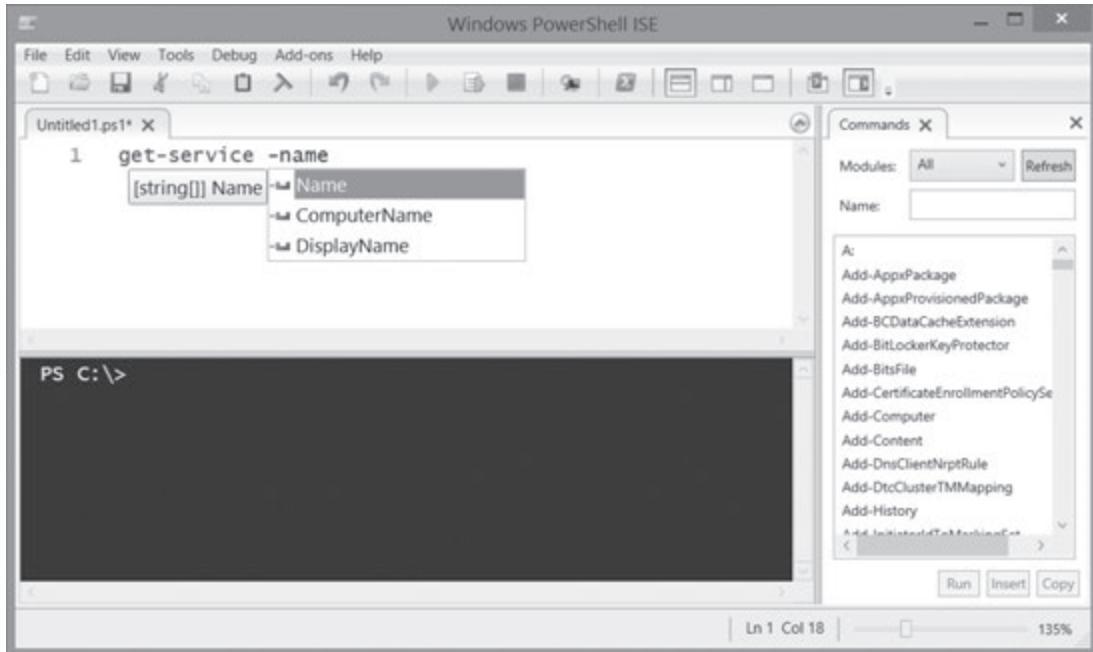


Figure 2-3

The default view of the ISE is divided into two resizable main sections and a **Commands** pane. The Help file contains information on all options, but let's review a few of the basics.

This scripting pane at the top can be resized to best suit your monitor and screen size. The scripting pane provides syntax coloring and IntelliSense when you type commands.

Below the scripting pane is the enhanced console pane. It also provides syntax coloring and IntelliSense.

The toolbar contains buttons that change the two pane view to a full screen view or a side by side view for multiple monitors. You should experiment to decide which view you prefer.

To the right is the **Command** pane and it is displayed by default. This pane can be closed and reopened to improve screen real estate. The **Command** pane lets you search for PowerShell commands, browse commands, and view command parameters.

In the lower right corner is a slider that will adjust the font size to suit your preferences. The ISE includes this sliding bar to quickly adjust the font size while you are working. If you need to adjust the font style and more features, select **Tools**, and then click **Options**.

The built-in ISE is a great tool for admins who are just learning to use PowerShell. While it's not intended to replace more full-featured commercial applications, it's a great tool for learning how to script and automate in PowerShell.

The ISE is available practically everywhere you have PowerShell installed, except on Windows Server 2008 R2. For Windows Server 2008 R2, you must install the ISE. To install ISE, run the following commands:

```
Import-Module ServerManager
Add-WindowsFeature PowerShell-ISE
```

---

**Note:**

Do not run the above on Server Core for Windows Server 2012 R2 unless you want the ISE and the Minimum Server Interface GUI.

The Server Core versions of Windows Server 2012 R2 cannot run the ISE (or any other professional editor), with the exception of Notepad. There are no graphical desktop components to support the program's ability to draw on the screen. This is a benefit to Server Core, and it is also a benefit to you.

We recommend that you script/automate on your client computer, not a server. Later, we will show you how you can edit scripts that are located on the server, from your client computer, so you really don't need an editor on the server.

The other concept here is that you really shouldn't work directly on a server desktop anyway. We know that Microsoft promoted the use of Remote Desktop Protocol (RDP) in the past, but this actually harms the server more than it helps. One of the reasons that Server Core is so stable and such a desired configuration is because it provides no graphical desktop to work on. You should install the Remote Server Administration Tools (RSAT) on your client. Then, you will have everything you need.

## Configuring SAPIEN's PrimalScript

Throughout this book you will notice examples and features from SAPIEN's PrimalScript and PowerShell Studio. Both of these tools are commercial scripting applications that share a common feature set and added special qualities for each product.

SAPIEN's PrimalScript is an enterprise level scripting platform designed to support the needs of the scripter/automator that works in a mixed environment. PrimalScript supports 50 scripting languages such as VBScript, REXX, Perl, JavaScript, .NET, and, of course, PowerShell. If you are responsible for scripting and automating in your existing environment, you probably already have a copy installed on your computer. If not, then download an evaluation copy and try it out as you go through this book. Many examples in this book were built using PrimalScript and you'll have a chance to see some of the special features that make PrimalScript the industry standard. While there is complete product documentation on using PrimalScript, here are some notes to get started quickly.

## Installing PrimalScript 2014

We are sure you've installed plenty of software in your time and PrimalScript is not going to be challenging for you.

After launching the downloadable executable, you can expect the standard set of questions regarding licensing, location, and program options. While you can customize the installation, the defaults should suffice for most people.

## Getting familiar with PrimalScript

After the installation completes, launch PrimalScript from the SAPIEN Technologies folder and pin it to your Taskbar, as you will want to open this often.

If this is your first time working with a professional scripting/development environment, the interface can seem daunting. Don't let that slow you down.

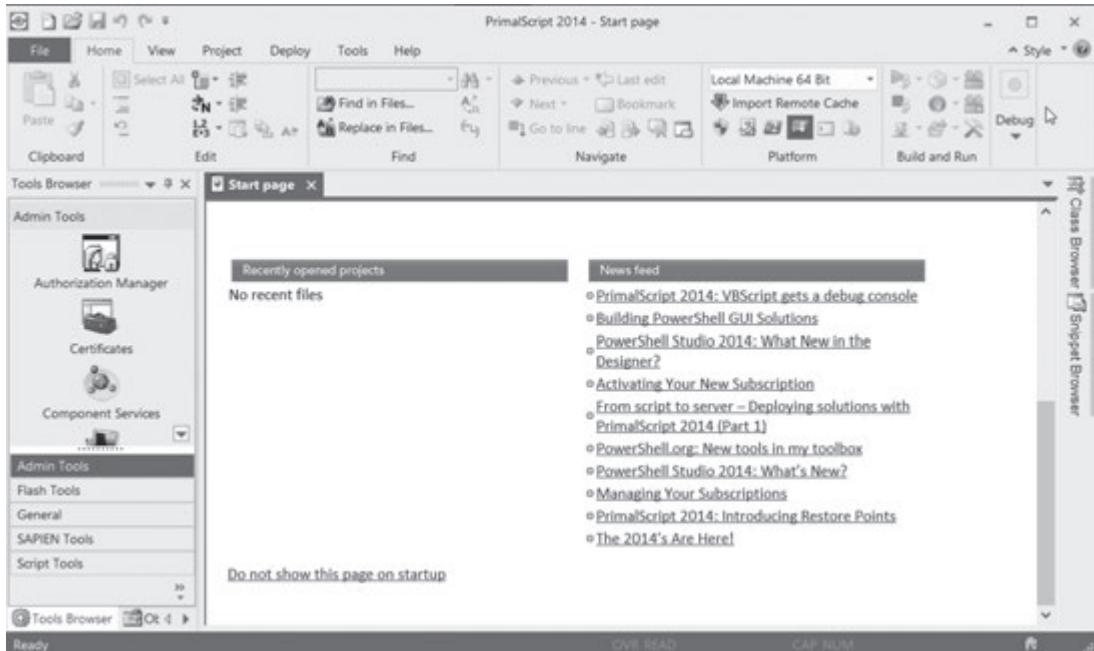


Figure 2-4

A quick look of the components:

- **Ribbon**—PrimalScript 2014 has added the ribbon and it's your quick action bar. When you're ready to save, run, or debug a script, this is where you will find everything you need.
- **Side panels**—Think of side panels as quick information assistants. While working on a script, you might need information from a tool such as Active Directory (AD), Users and Computers or the registry editor. At the bottom, you can select the **Object Browser** tab, which contains a consistent and easy to use display to help you navigate and find objects for your scripts like COM, WMI, .NET, and PowerShell (including cmdlets, aliases and, modules).

**Tip:**

Most admins prefer to unpin this side panel, to maximize the scripting page. The side panel will open when you hover your mouse over the tab.

- The **Snippet Browser** (located on the right) helps you quickly find pieces of code syntax, like templates, to assist in building loops and functions. Hover over the **Snippet Browser** and select the snippet you need.

---

**Tip:**

You can add your own snippets!

- The **Start Page** is a great place to locate recently opened scripts and projects, plus the latest news from SAPIEN Technologies. This tabbed view is the same location where your working scripts will be displayed.

You can customize this default view in a variety of ways to fit your own personal scripting experience. Many of these you will discover on your own, some will be demonstrated later in this book. There are just a couple that you need now.

## Files and Fonts

Like with the other tools, you will probably want to set the font size and colors to suit your own preferences. Selecting the **File** tab will display the standard options of **Open**, **Save**, and **New** that you expect from a **File** menu. Notice that towards the bottom of Figure 2-5, you will see the **Options** icon. This is where you can customize many settings for PrimalScript, including the fonts.

Windows PowerShell: TFM

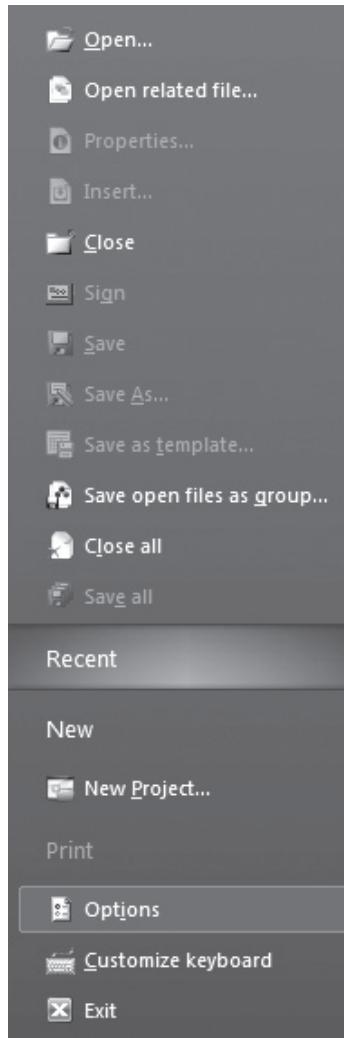


Figure 2-5

The **Options** icon opens the **Application/General** settings. Here, you can customize the color scheme and layout. The font settings are located under the **Text Editor/General** settings, as shown in Figure 2-6.

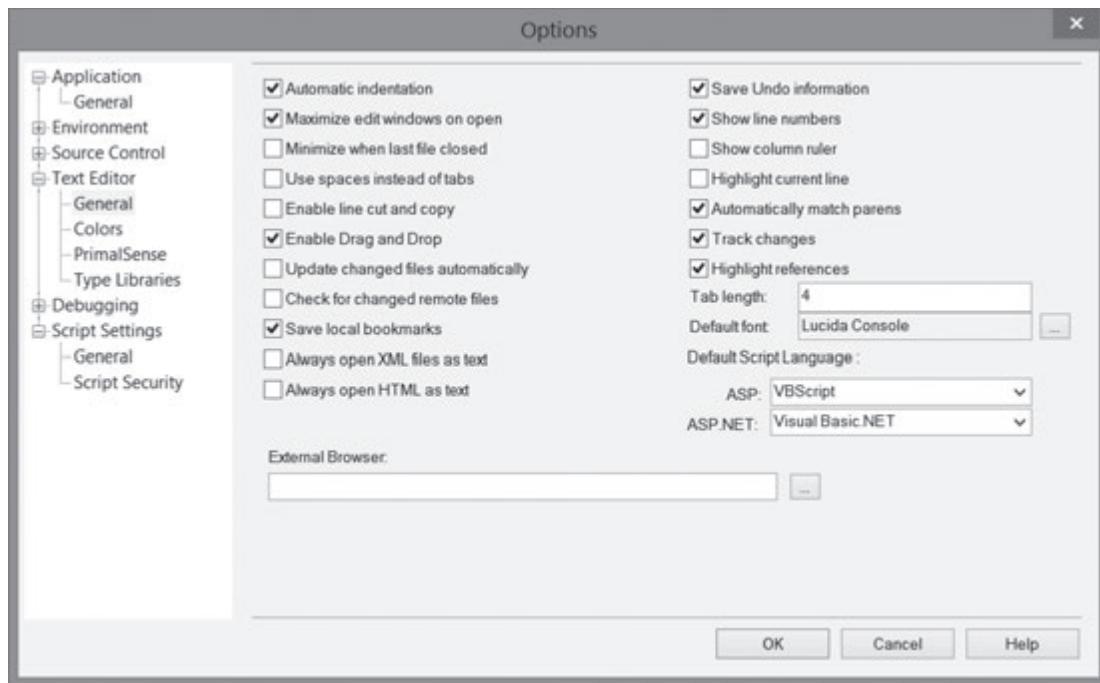


Figure 2-6

## Scripting with PrimalScript

PrimalScript includes a powerful scripting environment and a PowerShell console, so you can try commands before you copy them to your script. When you are ready, you will have many features to help you write your script, including:

- **Syntax highlighting**—Helps to visually indicate if you have a syntax mistake. The color coding of the text will change if there is a problem.
- **PrimalSense**—Notice the drop-down list for the command I'm typing and the pop-up Help for the syntax. PrimalScript will display this information while you type.

The screenshot shows the PrimalScript 2014 interface. The menu bar includes File, Home, View, Project, Deploy, Tools, and Help. The toolbar has various icons for file operations like Paste, Select All, Find in Files, Replace in Files, and Go to line. The main window shows a script named 'Untitled.ps1' in 'Global Scope'. The script content is:

```

1 $services = Get-Service | Where-Object {$_.Name -like "b*"} |
2 Select-Object -Property Name,Status | Format-Table -AutoSize
3 Write-Output $services

```

The output pane displays the results of the script execution:

BITS		Running
BrokerInfrastructure		Running
Browser		Running
bthserv		Running

Execution time: < 1 second

Figure 2-7

PrimalScript is the perfect enterprise scripting/automation platform. It provides features and assistance beyond the built-in PowerShell ISE. SAPIEN also created PowerShell Studio, which you will see throughout the book, and we will finish this chapter by taking a quick tour of that.

## Configuring SAPIEN's PowerShell Studio

SAPIEN's PowerShell Studio is a scripting/developing environment dedicated to PowerShell. Unlike PrimalScript and its 50 different scripting languages, PowerShell Studio is tightly focused on the PowerShell language. You will find many of the same great features, such as syntax coloring, PrimalSense, and a great debugger to help you build your scripts. PowerShell Studio includes a unique feature—the ability to create graphical Windows applications by using the PowerShell language. For the admin that needs to make tools in a graphical window for end users or a help desk, this is your product.

## Installing PowerShell Studio

We are sure you've installed plenty of software in your time and PowerShell Studio is not going to be challenging for you.

After launching the downloadable executable, you can expect the standard set of questions regarding licensing, location, and program options. While you can customize the installation, the defaults should suffice for most people.

## Getting familiar with PowerShell Studio

After the installation completes, launch PowerShell Studio from the SAPIEN Technologies folder. This is also a good time to pin it to your Taskbar, as you will want to open this often.

If this is your first time working with a professional scripting/development environment, the interface can seem daunting at first, but don't let that slow you down. Take a look at the general layout in Figure 2-8.

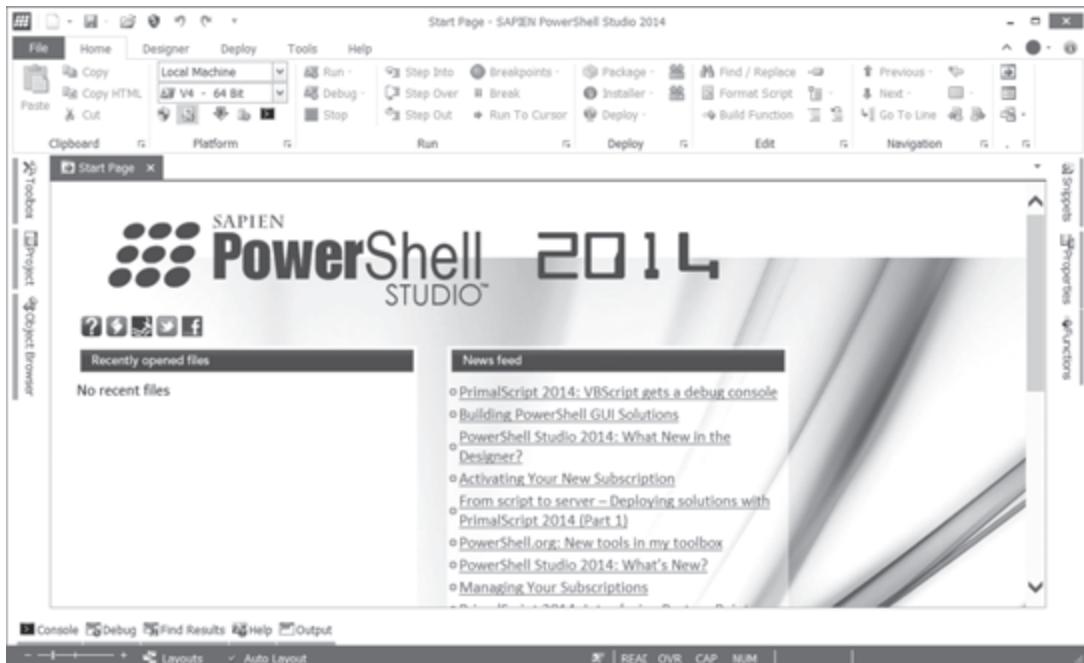


Figure 2-8

Here's a quick look at the components:

- The **Ribbon** is your action bar. When you're ready to save, run, or debug a script, this is where you will find everything you need.
- The **Start Page** is a great place to locate recently opened scripts and projects, plus the latest news from SAPIEN. This tabbed view is the same location where your working scripts will be displayed.
- **Side panels**—I think of side panels as quick information assistants. While working on a script, you might need help with a cmdlet or an example of the advanced function syntax. The side panels display information while you work and can be pinned (always open) or unpinned (the panel is hidden until you hover your mouse over it).
- **Output** pane—The default layout displays an output pane below your script so that you can see the results of your work when you run the script. Notice that this is tabbed with several views, such as **Console**.
- Several default **Layouts** are included so that you can quickly shift the entire layout (panels and all) to a different view.

On the **Home** ribbon bar, you will notice you can click on **File**, then **Options**. This allows you to customize PowerShell Studio, including the fonts.

## Setting the fonts for comfort

Click **File**, **Options**, and then the **Editor** tab—this is where you can customize the **Font and Color** settings as shown in Figure 2-9.

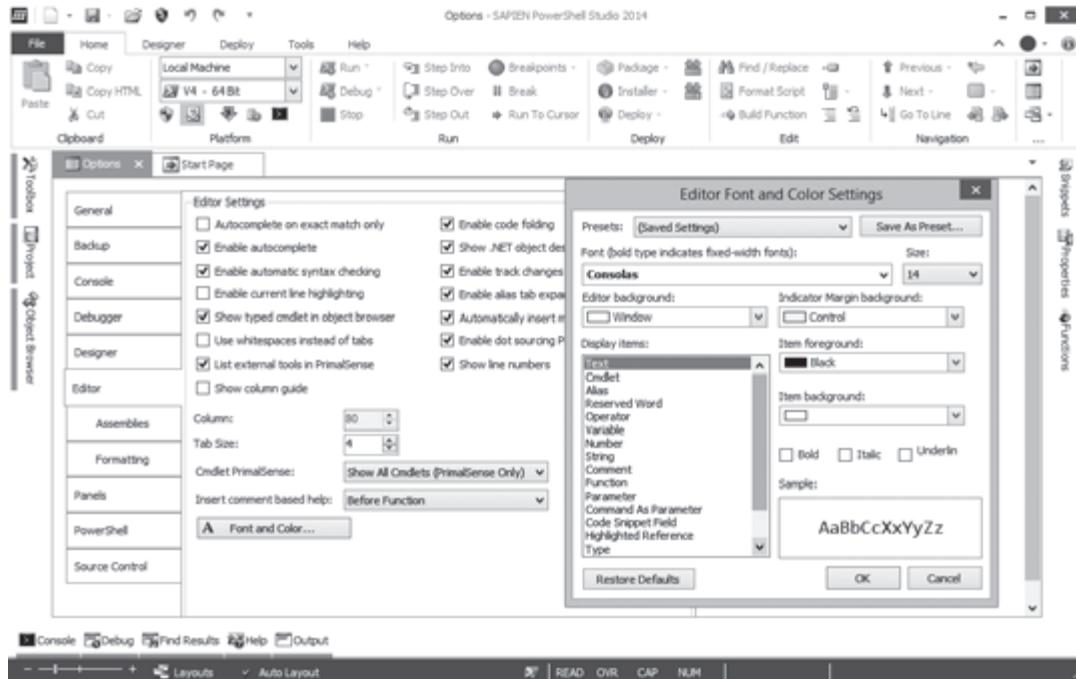


Figure 2-9

Before you leave the **Options** window, you might want to adjust the font for the built-in PowerShell console. The default is already set to Lucida Console, however depending on your monitor size and resolution, you should adjust the size.

## Scripting with PowerShell Studio

Several chapters in this book focus on creating a Windows application using PowerShell Studio, but you can always start scripting now with PowerShell Studio by using the PowerShell editor pane.

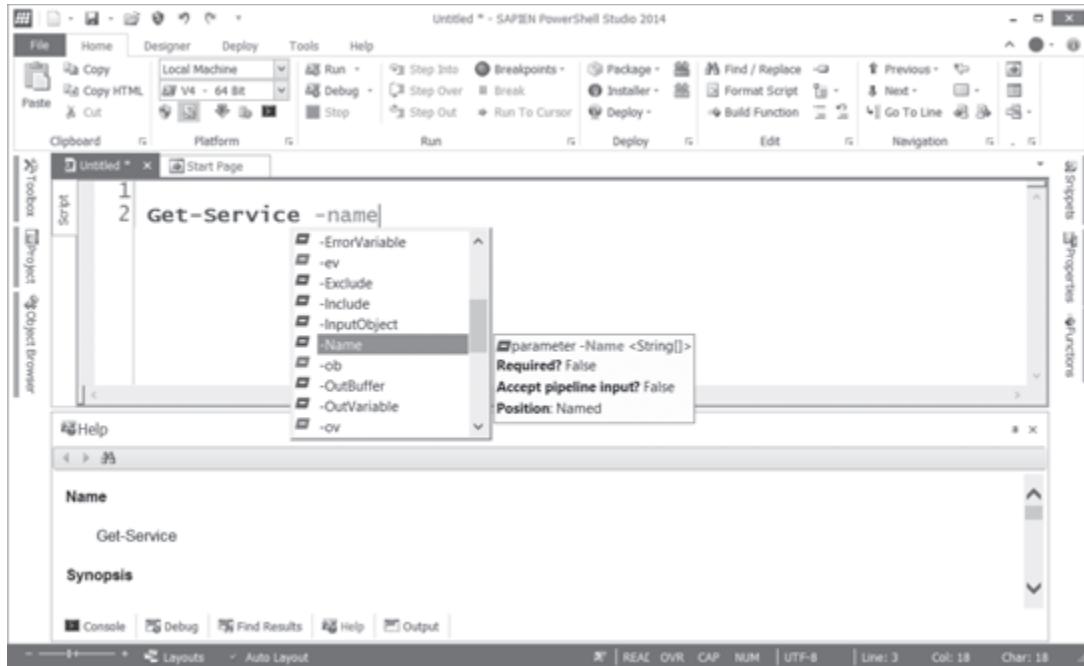


Figure 2-10

You have great features to help you get started, such as the following:

- **Syntax highlighting**—Helps to visually indicate if you have a syntax mistake. The color-coding of the text will change if there is a problem.
- **PrimalSense**—Notice the drop-down list for the command that is in the process of being typed into Studio. PowerShell Studio will display this information while you type, providing Help along the way!

With all this information about the tools, Windows PowerShell console, Windows PowerShell ISE, SAPiEN's PrimalScript, and PowerShell Studio, it's time to set them up and start working. Try the following exercise to make sure you have everything you need and that it is configured.

## Exercise 2 – Preparing your environment

It's time to install the current version of PowerShell on your computer, including the additional tools discussed in this chapter. If you already built a VM of Windows Server 2012 R2 or Windows 8.1, you are ready to go. If you haven't, now is the time to update your computer to the latest PowerShell version or build that VM!

### Task 1

Make sure that you have the current version of PowerShell installed and ready to go on your laptop, desktop, or the extended lab environment. You should pin the PowerShell console icon to your Taskbar. Remember: on 64-bit systems, make sure to pin the **x64** PowerShell icon.

You should launch the console and verify that **Administrator** is displayed on the menu bar. If not, right-click the PowerShell icon and select **Run As Administrator**.

### Task 2

Configure the PowerShell console with the fonts and font sizes that work best for you. You can configure the color of the shell if you like as well.

### Task 3

Locate the PowerShell ISE and pin it to your Taskbar. You should take a few minutes and customize your environment. Remember to try out the different screen options.

### Task 4

You should install the trial version of SAPIEN PowerShell Studio and pin it to your Taskbar. You will use this application later when creating Windows applications. Configure the font style and size to your liking.

### Task 5

You should install the trial version of SAPIEN PrimalScript and pin it to your Taskbar. Configure the font style and size to your liking.

## Chapter 3

# Don't fear the shell

### Familiar commands

Many admins mistakenly don't launch and use PowerShell. They believe that PowerShell is only a scripting language, like VBScript, when in fact it's far from that. While PowerShell does include a scripting language, the console is designed to be a powerful interactive command console, allowing admins to interact with their environment in real time.

This means that PowerShell is designed to run commands, many you are already familiar with, along with new types of commands called cmdlets. For this reason, there is really no need to open an old command prompt—PowerShell runs the Windows native commands like a pro!

```
IPConfig /all  
Ping localhost
```

These commands are not PowerShell versions—they are the same commands you have used for years and they provide the same results. PowerShell knows how to execute them.

PowerShell can launch Windows programs such as Notepad, MSPaint, or whatever standard executable that you usually use with the older command prompt.

```
notepad  
calc  
mspaint
```

### Windows PowerShell: TFM

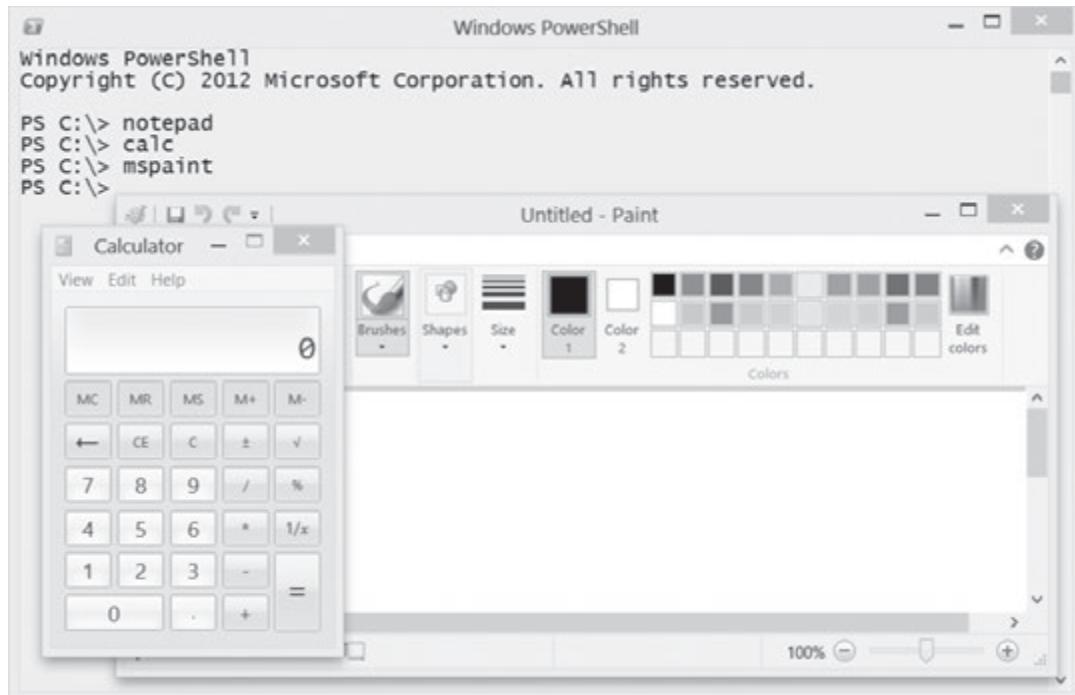


Figure 3-1

You even have familiar navigation commands for the file system such as **Dir**, **Md**, **Cd**, **Rd**, as well as the \*nix style **Mkdir**, **Ls**, and others. These commands are not executables and you might wonder “how does PowerShell know how to execute these?” The answer is simple: PowerShell isn’t running the old DOS versions of these, PowerShell is running an alias (think shortcut) that points to its own commands and they have a similar function and output. In other words, when you type **Dir**, which is an alias, PowerShell is actually running **Get-ChildItem**.

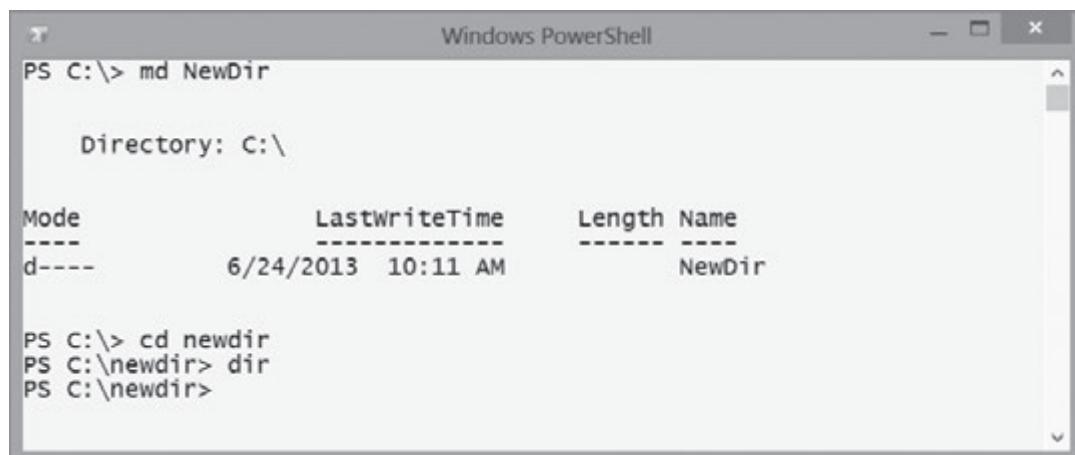


Figure 3-2

These shortcuts—old-style commands that point to the new PowerShell cmdlets—are known as Aliases. Keep in mind that aliases point to PowerShell commands, not the old-fashioned CMD.exe ones. This can cause problems—for example, **Dir \*.exe /s** doesn't work. That's the old command. Now, you need to use the parameters for the PowerShell cmdlet—**Get-ChildItem -Filter \*.\***—**-Recurse**. Don't worry though, we will show you all about PowerShell cmdlet parameters and arguments in Chapter 4. For now, keep in mind that PowerShell is designed to be used as the old command shell replacement and make it easy for you to convert to the new shell.

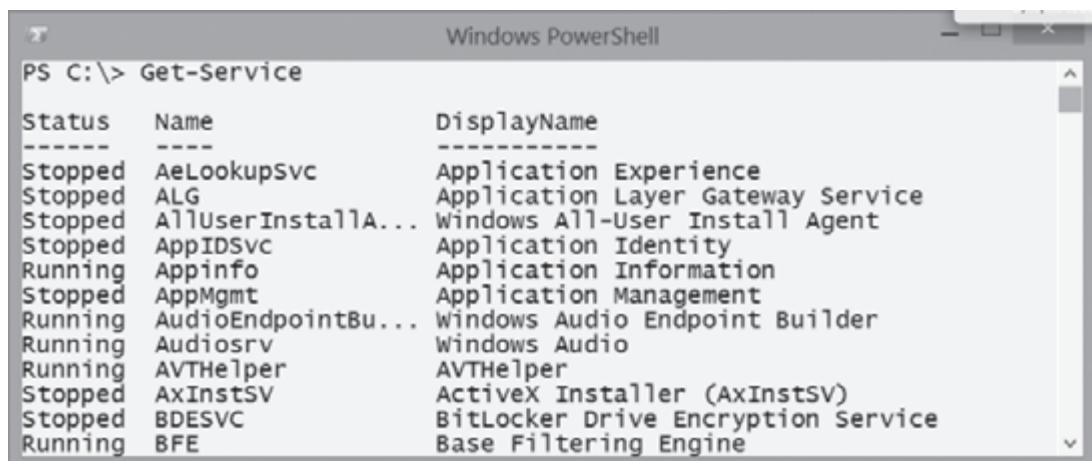
We recommend that you use PowerShell for all your command needs, even tell other admins and demonstrate the above commands to them. Start using PowerShell, even if you don't know the new cmdlets yet. It will help alleviate some of the fear in learning to use PowerShell. Also, because you will be working in PowerShell, you will start to learn the new commands gradually and begin to expand your control over your network.

## The new commands—cmdlets

The new commands that provide PowerShell with its power to manage products and provide automation are known as cmdlets (pronounced command-lets). The Microsoft PowerShell team wrote a series of core cmdlets, and then the Product teams added their own (usually in Modules, which are discussed later). If you keep working through the book, you will also create your own cmdlets for PowerShell!

At first these commands seem unusual in that they are named with a verb-noun combination. It turns out that this is very helpful in determining what a command will do—and what it will perform its action on.

As an example, think of a good verb that represents retrieving information. I would guess you said “get” and that means you’re on the right path. If you want to “get” information about the services running on your computer, then think of a noun that would best represent “services.” Nouns are singular in PowerShell, so the best noun would be “service.”



```
Windows PowerShell
PS C:\> Get-Service
Status   Name           DisplayName
-----   -- --          -----
Stopped  AeLookupSvc    Application Experience
Stopped  ALG            Application Layer Gateway Service
Stopped  AllUserInstallA... Windows All-User Install Agent
Stopped  AppIDSVC       Application Identity
Running   Appinfo        Application Information
Stopped  AppMgmt        Application Management
Running   AudioEndpointBu... Windows Audio Endpoint Builder
Running   Audiosrv       Windows Audio
Running   AVTHelper      AVTHelper
Stopped  AxInstSV       ActiveX Installer (AxInstSV)
Stopped  BDESVC         BitLocker Drive Encryption Service
Running   BFE            Base Filtering Engine
```

Figure 3-3

PowerShell cmdlets connect the verb to the noun with a (-) dash. A cmdlet that will retrieve information about your services is **Get-Service**.

Don't worry, you don't need to guess every time you want to find a cmdlet. In fact, as you will see shortly, PowerShell will help you find them. You don't even have to type the full cmdlet name, as many cmdlets have aliases (shortcuts) to reduce your typing, similar to **Dir**.

The point to keep in mind is that cmdlets are designed for you to understand their purpose and use them easily. Think of it this way: "What do you want to do?" as the verb and "What do you want to do it to?" as the noun.

You can view a list of all verbs using the cmdlet **Get-Verb**.

Verb	Group
Add	Common
Clear	Common
Close	Common
Copy	Common
Enter	Common
Exit	Common
Find	Common
Format	Common
Get	Common
Hide	Common
Join	Common
Lock	Common
Move	Common
Remove	Common
Set	Common
Stop	Common

Figure 3-4

Don't try to memorize this list. You will quickly become very comfortable with the most common verbs and the nouns that go along with them.

## Aliases, the good and the bad

Aliases are basically just nicknames or shortcuts for cmdlets, either to make it easier to type the cmdlet name or to associate the cmdlet with a familiar command from the older Cmd.exe shell. After all, it's certainly easier to type **Dir** than to type **Get-ChildItem** all the time, and **Dir** corresponds with a Cmd.exe command that performs basically the same function as **Get-ChildItem**. PowerShell comes with a number of predefined aliases that make typing faster. Simply run **Get-Alias** to see a complete list of all aliases, as well as the cmdlets that they point to.

CommandType	Name	ModuleName
Alias	% -> ForEach-Object	
Alias	? -> Where-Object	
Alias	ac -> Add-Content	
Alias	asnp -> Add-PSSnapin	
Alias	cat -> Get-Content	
Alias	cd -> Set-Location	
Alias	chdir -> Set-Location	
Alias	clc -> Clear-Content	
Alias	clear -> Clear-Host	
Alias	clhy -> Clear-History	
Alias	cli -> Clear-Item	
Alias	clp -> Clear-ItemProperty	
Alias	cls -> Clear-Host	
Alias	clv -> Clear-Variable	

Figure 3-5

You can make your own aliases, too. For example, it's useful to occasionally open Windows Notepad to jot down a few notes as you're working in the shell, and simply typing Notepad over and over takes too long. Instead, it's easier to use a shorter alias, such as **Np**, which you can create by running the following:

```
New-Alias np notepad
```

Notice that you aliased an external command! Notepad isn't a PowerShell cmdlet, but it is something you can run in PowerShell. Therefore, you can create an alias for it. Of course, your alias will not be saved after you close the shell session. Later, we will show you how to add your aliases to a profile script, so that they are automatically loaded. For now, realize that you can use and create your own aliases.

## The downside to aliases

Aliases have some downsides. For one, while they're certainly easier to type, they can be harder to read.

```
ps | ? { $_.CPU -gt 50 } | % { $_.Name }
```

Yikes. Even punctuation marks like **?** and **%** can be used as aliases! This is a lot easier to figure out when full cmdlet names are used.

```
Get-Process | Where-Object { $_.CPU -gt 50 } | ForEach-Object { $_.Name }
```

While there are still plenty of strange symbols in the previous example, at least using the full cmdlet names help you to understand what the command will produce. This command retrieves all currently running processes, selects those instances that have CPU utilization greater than 50, and then for each of those instances, just displays the process name.

Aliases are limited to providing a shorter, alternate name, only. You cannot create an alias for an expression that includes a parameter.

```
New-Alias -name IP -value IPCConfig /all
```

If you try to create an alias like this, you will receive an error.

```
New-Alias : A parameter cannot be found that matches parameter name '/all'.
```

```
At line:1 char:10
```

This is not to say that you can't create an alias called IP that points to **IPConfig**, you just can't include any additional parameters.

Another downside to aliases is that unless you stick to the aliases predefined in PowerShell itself, any scripts that you write won't run on another computer unless you first take the time to define your custom aliases on that computer. We recommend that you avoid using aliases when you are scripting, even the predefined ones. It just makes the script too hard to read in the future.

When it comes to writing scripts, you can work around both of these downsides by using a professional script editor like SAPIEN PrimalScript or PowerShell Studio.

---

**Note:**

Type all the aliases you want—after all, they are faster—and then go to PrimalScript's Edit menu. Click Convert, and then select Alias to Cmdlet to have PrimalScript expand all of your aliases into their full cmdlet names. You'll have instant readability and portability!

The screenshot shows the PrimalScript 2014 interface. The main window displays a PowerShell script:

```

1
2
3
4
5
6
7
8
9
10
11 $ps | ? { $_.CPU -gt 50 } | % { $_.Name }
12 Get-Process | Where-Object { $_.CPU -gt 50 } | ForEach-Object { $_.N
13

```

The script filters processes by CPU usage and then lists their names. A context menu is open over the second line of the script, showing options like Uppercase, Lowercase, and various encoding and conversion tools.

Figure 3-6

There are a couple of other cmdlets that you can use to work with aliases. **Export-Alias** exports your aliases into a special file, allowing you to import those aliases to another system (using **Import-Alias**). The **Set-Alias** cmdlet lets you change an existing alias. You will read more about using these cmdlets in Chapter 4, and a lot more, when we show how to use the Help system.

## Command-line assistance

As you start to work with PowerShell at the console, you will start to discover that PowerShell is trying to help you build the commands that are so important to managing your network environment. Here is a list of the most common command-line assistance tools.

### Command History

Like Cmd.exe and most other command-line environments, PowerShell maintains a history (buffer) of commands that you've typed (this is different from the command history that the **Get-History** and **Add-History** cmdlets can manipulate). Pressing the Up and Down arrow keys on your keyboard provides access to this history, recalling past commands so that you can either easily run them again or allowing you to quickly change a previous command and run a new version.

A little known shortcut is the **F7** key, which pops up a command history window, allowing you to scroll through the window and select the command you want. If you press **Enter**, the selected command executes immediately. If you press the right or left arrow key, the command will be inserted but not executed. This is helpful when you want to recall a complicated command, but need to tweak it. Press **Esc** to close the window without choosing a command.

```
PS C:\> Get-History
Id CommandLine
-- -----
4 Clear-History
5 Get-Service
6 Get-Process
7 get-eventlog -
8 cls

PS C:\>
```

Figure 3-7

To increase the command buffer, right-click the **System** menu of your PowerShell window and then select **Properties**. On the **Options** tab, you can increase the buffer size from the default of 50.

Finally, one last trick for the command history: type the beginning of a command that you've recently used, and then press **F8**. PowerShell will fill in the rest of the command and you can press **F8** again to find the next match to what you've typed. This is a quick way to recall a command more conveniently than by using the arrow keys.

### **Another history—Get-History**

Another way of seeing your command-line history is to use the **Get-History** cmdlet. **Get-History** maintains your command history for the last 4096 entries of your current session, as opposed to PowerShell version 2 that only maintained 32.

```
PS C:\> Get-History
Id CommandLine
-- -----
4 Clear-History
5 Get-Service
6 Get-Process
7 get-eventlog -LogName system -Newest 5 -EntryType error
8 Get-History

PS C:\>
```

Figure 3-8

WOW! That's a lot of history. Later you will learn how you can export your history to a text file, CSV, or XML file. The number can be increased by setting the preference variable **\$MaximumHistoryCount**—but we haven't talked about variables yet, so the default should be good enough for now.

## Line editing

While PowerShell doesn't provide a full-screen editor (if you need that, then it may be time to investigate a visual development environment that supports PowerShell, such as SAPIEN PowerShell Studio—[www.SAPIEN.com](http://www.SAPIEN.com)), it does provide basic editing capabilities for the current line.

- The Left and Right arrow keys move the cursor left and right on the current line.
- Pressing Ctrl+Left arrow and Ctrl+Right arrow moves the cursor left and right one word at a time, just like in Microsoft Word.
- The Home and End keys move the cursor to the beginning and end of the current line, respectively.
- The Insert key toggles between the insert and overwrite modes.
- The Delete key deletes the character under the cursor.
- The Backspace key deletes the character behind, or to the left of, the cursor.
- The Esc key clears the current line.

## Copy and paste

If you've enabled QuickEdit and Insert Mode for your PowerShell window, you can easily copy and paste between PowerShell and Windows. To copy from PowerShell, merely use the mouse to select the text, and then press **Enter**. You can then paste it into another application like Notepad. To paste something into PowerShell, position your cursor in the PowerShell window and right-click. The copied text will be inserted at the command prompt.

## Tab completion

Also called command completion, this feature exists to help you complete command names and even parameter values more quickly. Pressing **Tab** on a blank line will insert an actual tab character; any other time—if you've already typed something on the line, that is—the **Tab** key kicks into command-completion mode. Here's how it works:

- If you've just typed a period, then command completion will cycle through the properties and methods of whatever object is to the left of the period. For example, if `$wmi` represents a WMI object, typing `$wmi.` and pressing **Tab** will call up the first property or method from that WMI object.
- If you've typed a period and one or more letters, pressing **Tab** will cycle through the properties and methods that match what you've already typed. For example, if `$wmi` is an instance of the `Win32_OperatingSystem` WMI class, typing `$wmi.re` and pressing **Tab** will display “`$wmi.Reboot()`”, which is the first property or method that begins with “re”.
- If you've just typed a few characters, and no punctuation other than a hyphen (“-”), command completion will cycle through matching cmdlet names. For example, typing `get-w` and pressing **Tab** will result in `Get-WmiObject`, the first cmdlet that begins with “get-w”. Note that this does not work for aliases, only cmdlet names.

- If you've typed a command and a partial parameter name, you can press **Tab** to cycle through matching parameter names. For example, you can type **gwmi -comp** and press **Tab** to get **gwmi -computerName**.
- If none of the above conditions are true, then PowerShell will default to cycling through file and folder names. For example, type **cd**, a space, and press **Tab**. You will see the first folder or file name in the current folder. Type a partial name, like **cd Doc**, and press **Tab** to have PowerShell cycle through matching file and folder names. Wildcards work, too. For example, type **cd pro\*files** and press **Tab** to get **cd Program Files**.

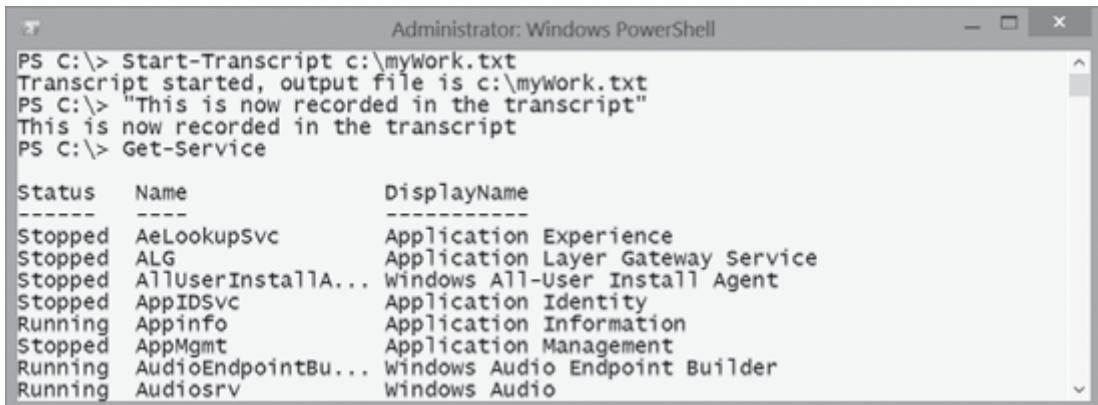
Those are the basics of working with the PowerShell console, but there is one more feature that we love the best—Transcripts.

## Keeping a transcript

Sometimes, it is useful to keep track of exactly what you're doing in PowerShell. For example, if you're experimenting a bit, going back and reviewing your work can help you spot things that worked correctly. You could then copy and paste those command lines into a script for future use. It's good to have a record of both the commands and the output—a complete transcript of everything you do.

PowerShell offers a way to keep track of your work through a transcript. You start a new one by using the **Start-Transcript** cmdlet.

```
Start-Transcript C:\myWork.txt
```



The screenshot shows an Administrator: Windows PowerShell window. The command `Start-Transcript c:\myWork.txt` is run, followed by the message "Transcript started, output file is c:\mywork.txt". Then, the command `"This is now recorded in the transcript"` is run twice. Finally, the command `Get-Service` is run, displaying a table of services:

Status	Name	DisplayName
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Stopped	AllUserInstallA...	Windows All-User Install Agent
Stopped	AppIDSvc	Application Identity
Running	Appinfo	Application Information
Stopped	AppMgmt	Application Management
Running	AudioEndpointBu...	Windows Audio Endpoint Builder
Running	Audiosrv	Windows Audio

Figure 3-9

A transcript is just a simple text file that contains everything shown in the PowerShell console window. A downside to it is that if you want to copy and paste your work into a script, you first have to edit out all the command prompts, your mistakes, and so forth. Once started, a transcript will continue recording your work until you either close the shell or run the **Stop-Transcript** cmdlet.

```
Stop-Transcript
```

The **Start-Transcript** cmdlet has additional parameters (which you can look up by using Help) that append to an existing file, force an existing file to be overwritten, and so forth.

To view your transcript, you can use any text editor, such as Notepad.

```
notepad c:\MyWork.txt
```

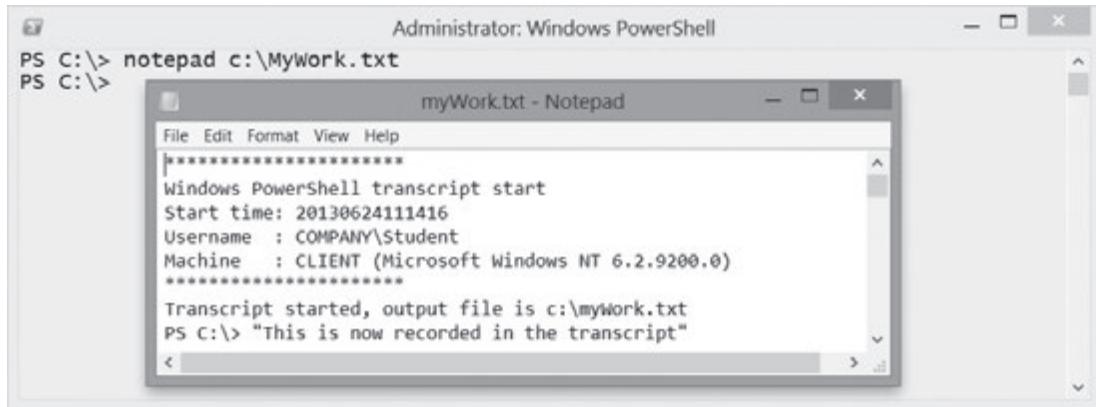


Figure 3-10

There is one drawback to **Start-Transcript**—it doesn't work in the ISE.

---

**Note:**

There are other DOS commands that don't work either. For a list, check the variable **\$psUnsupportedConsoleApplications** in the ISE.

So, if you want to record everything you are typing in the current PowerShell session, you need to run this from the PowerShell console.

## Exercise 3 – Don’t fear the shell

Time: 20 minutes

In this exercise, the goal is to try a variety of commands native to Windows, along with the aliases built into PowerShell. If you haven’t used some of these commands before, don’t worry—you don’t need to complete this exercise in order to be successful. This is just a chance for you to experiment with PowerShell and realize that it’s a complete replacement for the older cmd.exe console. You’ll use some familiar basic commands to manipulate files and folders. Here’s a reminder of some of the commands you’ll need:

- Dir, Ls
- Redirect a command’s output to a text file, as in Dir > list.txt
- Copy, Cp
- Type, Cat
- Ren
- Mkdir
- Copy, Cp
- Del, Rm

If you aren’t sure how to use some of these commands, again, don’t worry—this book will take you through all of them, from a PowerShell perspective.

### Task 1

Start a transcript for this exercise. Save the transcript to your Documents folder or, if you have permissions, the root of C:\. Name the transcript Exercise3.txt.

### Task 2

Create a text file in the root of C:\ that contains the names of the files and folders in C:\Windows\system32\WindowsPowerShell\V1.0. Name the text file MyDir.txt.

### Task 3

Can you display the contents of that text file?

Remember the old **Type** command?

## Task 4

Rename the file from MyDir.txt to PowerShellDir.txt.

## Task 5

Create a new folder named ExerciseOutput. You can either do this in your Documents folder or in the root of your C:\ drive.

## Task 6

Copy PowerShellDir.txt into the ExerciseOutput folder.

## Task 7

List your current command history.

## Task 8

Ping your localhost.

## Task 9

Retrieve the current IP information on your host, by using IPConfig.

## Task 10

Can you redirect your command history to a text file C:\history.txt?

## Task 11

Find a list of verbs that are used by the cmdlets.

## Task 12

Find a list of services or processes.

## Task 13

Stop your transcript and examine the results in Notepad.



## Chapter 4

# Finding Help when needed

### The extensive and evolving Help system

The key to learning and using PowerShell is becoming an expert at using the Help system. Read that again. Working with a GUI-based tool, you are guided through solving your problem. The problem with this is that you're guided through a series of steps envisioned by the development team that made the tool. What do you do if your business problem is not handled by the GUI tool? You move to a tool (PowerShell) that provides complete flexibility—off the rails—so that you can solve your problem.

The challenge that admins face with this complete freedom is finding the commands they need, how they work, and stringing them all together. The reason that command-line interfaces were so hard to use in the past was because of a lack of discoverability and incomplete documentation. PowerShell solves this with an extensive Help system that is easy to search—think of Google or Bing, but only focused on PowerShell—and complete documentation on syntax, parameter definitions, and examples of how to use the commands.

The Help system expands as you add modules that contain more cmdlets, so that you always have the ability to find what you need. Think of this: there are thousands of cmdlets. How many of those are you going to remember? We remember only one—and that's HELP!

## Updating Help

Before we start showing you how to use the Help system, one of the first tasks you need to accomplish is to ensure that you have the latest version of Help. In older versions of PowerShell, the Help system was static, meaning that it couldn't be updated. Even the best humans make mistakes and sometimes there were errors in the Help system that could lead you astray. Starting with PowerShell version 3, the Help system can be updated at any time.

`Update-Help`

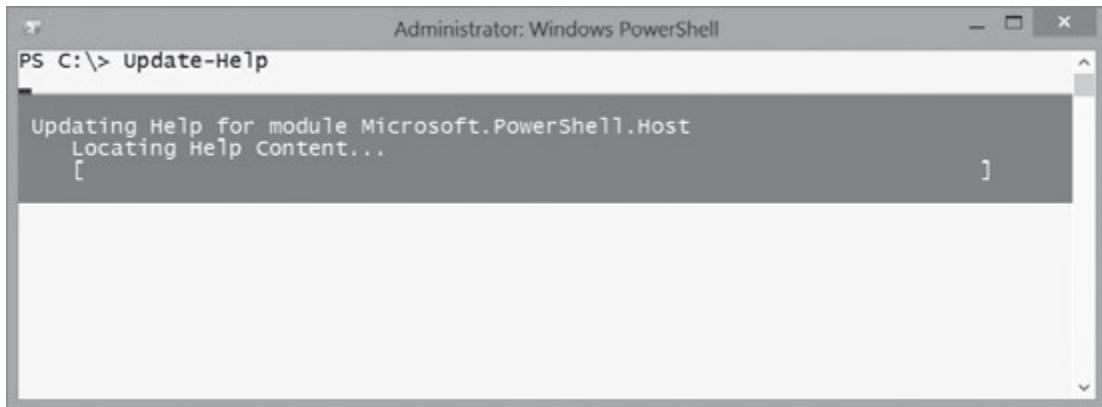


Figure 4-1

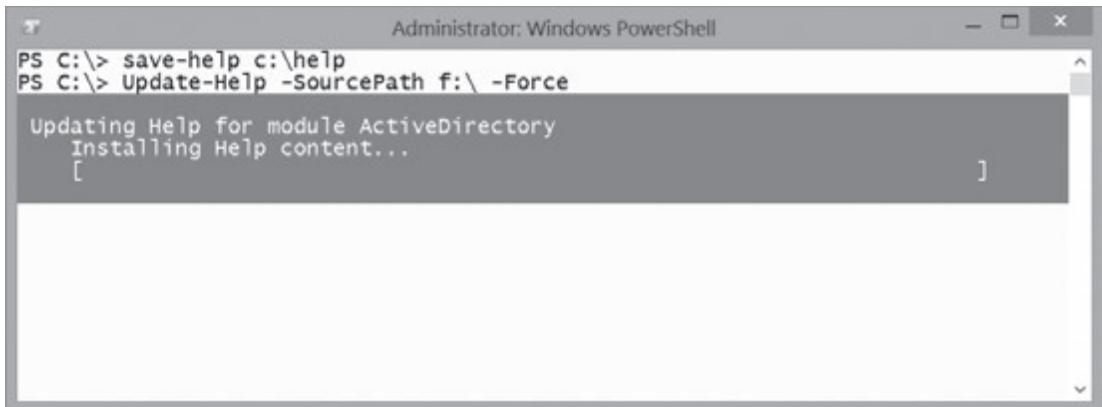
The **Update-Help** cmdlet checks the current version of your local Help files and if they are out of date, it downloads the latest version. If you're missing Help files for a newly installed module (additional cmdlets), it will update those as well.

Your computer must be connected to the Internet to receive the download. This might not be practical for all computers that you want to download the Help files for, so you can download the files to a directory location, and then share them out to local computers that do not have Internet access. On the computer with Internet access, type the following:

```
save-help c:\help
```

After sharing the folder, other computers can connect to the share. To share the folder, type:

```
Update-Help -SourcePath f:\ -Force
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> save-help c:\help is entered, followed by PS C:\> Update-Help -SourcePath f:\ -Force. The output shows "Updating Help for module ActiveDirectory" and "Installing Help content...". A progress bar is visible at the bottom of the console window.

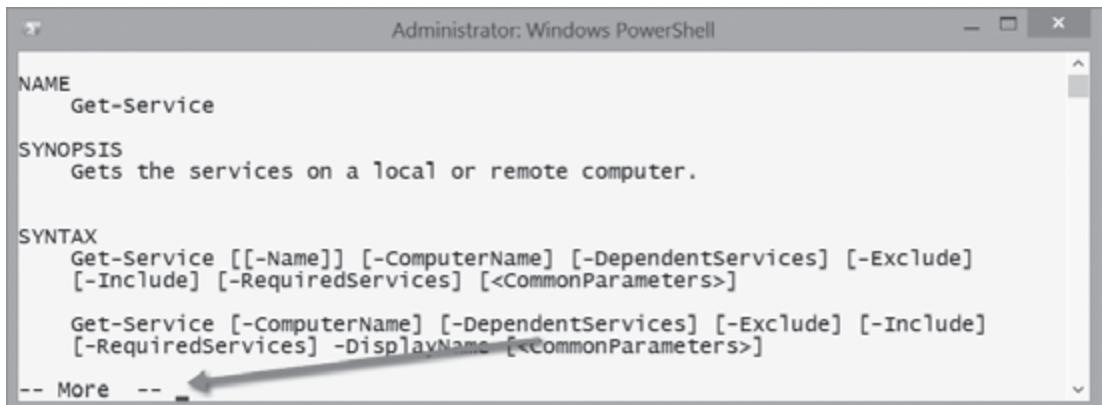
Figure 4-2

There are more options when using **Update-Help** and you can discover those shortly after we show you how to use the Help system you just updated!

### Discovering the commands you need

Microsoft ships PowerShell with extensive Help for all of the built-in cmdlets. When you need to access Help, you can use the cmdlet **Get-Help** or the special function named **Help** or the alias **Man**. When you use **Get-Help**, the Help content scrolls in the console to the end of the Help file—you can scroll back to read the parts of the file you want.

Some people prefer to page through the content, one page at a time, which is what happens when you use the special functions **Help** or **Man**.



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Get-Help Get-Service is entered. The output shows the NAME section with "Get-Service", the SYNOPSIS section with "Gets the services on a local or remote computer.", and the SYNTAX section with two examples of the Get-Service cmdlet. An arrow points from the text "More" at the bottom left to the scroll bar on the right side of the console window.

Figure 4-3

Other than paging, there is no difference in using these three commands to access Help and you will see them used interchangeably in this book.

## Discovering commands

The **Help** commands can search the entire Help system to find the cmdlets and documentation that you need. Think of **Get-Help** as a search engine focused on finding PowerShell results for you. As an example, say you wanted to find out if PowerShell had cmdlets to help you work with the processes on your computer.

```
Get-Help *Process*
```

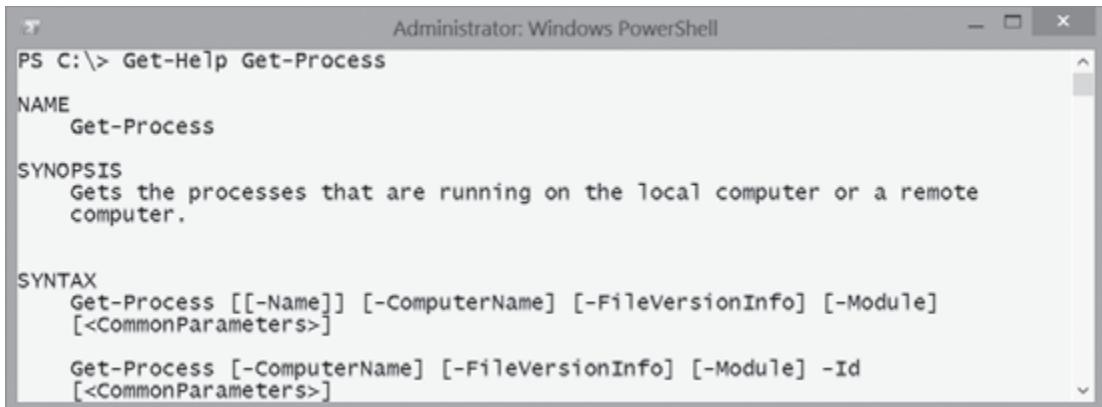
The wildcard characters (\*) tell the Help system to look for anything that contains the word "process", even if its proceeded or suffixed by other words. See? A lot like using Google or Bing—just focused on PowerShell.

Name	Category	Module	Synopsis
Debug-Process	Cmdlet	Microsoft.PowerShell.M...	Debugs...
Get-Process	Cmdlet	Microsoft.PowerShell.M...	Gets t...
Start-Process	Cmdlet	Microsoft.PowerShell.M...	Starts...
Stop-Process	Cmdlet	Microsoft.PowerShell.M...	Stops ...
Wait-Process	Cmdlet	Microsoft.PowerShell.M...	Waits ...

Figure 4.4

The **Help** command discovered five cmdlets that work with processes. If you wanted to see a list of processes, the cmdlet **Get-Process** might do the trick. To know for sure, and to see the cmdlet specific help, type **Help**, followed by the name of the cmdlet, without wildcards.

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



Administrator: Windows PowerShell

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

**NAME**

Get-Process

**SYNOPSIS**

Gets the processes that are running on the local computer or a remote computer.

**SYNTAX**

```
Get-Process [[-Name]] [-ComputerName] [-FileVersionInfo] [-Module]
[<CommonParameters>]
```

```
Get-Process [-ComputerName] [-FileVersionInfo] [-Module] -Id
[<CommonParameters>]
```

Figure 4-5

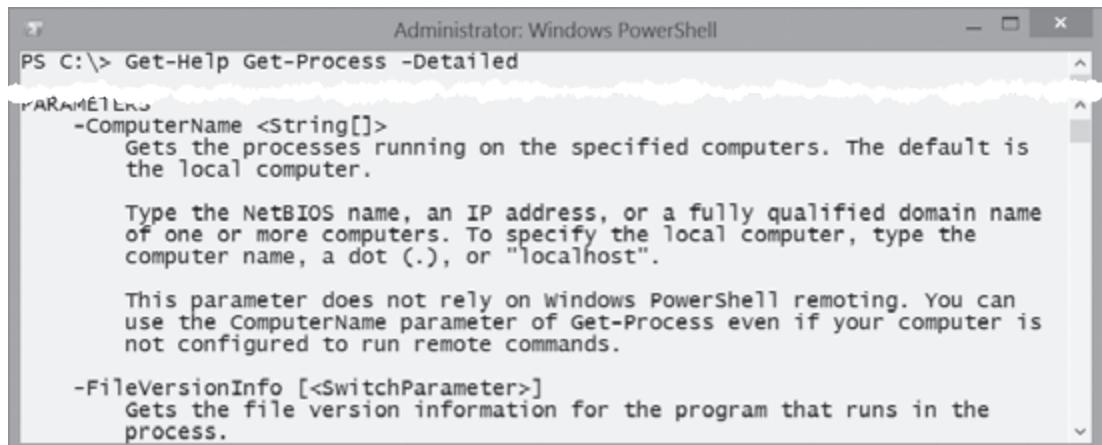
We will explain the components inside the cmdlet specific Help in a moment, but the basic layout will probably make sense. The basic Help file will show the name of the cmdlet, a brief description, the syntax, and a longer description. This is helpful, but there is more that the Help command can produce.

### Parameter switches for more Help

The Help commands support several parameter switches that can increase the amount of Help you receive, or dial into specific information. Shortly, we will show you how to read a cmdlet Help file, but these switches are important now.

The default Help provides syntax and description information about the cmdlet that you are examining. It doesn't provide all the details and examples that you will need as you work with cmdlets.

One of the more useful parameter switches for the Help commands is **-Detailed**. This provides much more information about the parameters needed for the syntax, including definitions and default settings.



Administrator: Windows PowerShell

```
PS C:\> Get-Help Get-Process -Detailed
```

**PARAMETERS**

**-ComputerName <String[]>**

Gets the processes running on the specified computers. The default is the local computer.

Type the NetBIOS name, an IP address, or a fully qualified domain name of one or more computers. To specify the local computer, type the computer name, a dot (.), or "localhost".

This parameter does not rely on Windows PowerShell remoting. You can use the ComputerName parameter of Get-Process even if your computer is not configured to run remote commands.

**-FileVersionInfo [<SwitchParameter>]**

Gets the file version information for the program that runs in the process.

Figure 4-6

The **-Detailed** switch also includes examples of how to use the cmdlet. While not every possible usage is outlined, the examples give numerous and useful real-world explanations.

The screenshot shows an Administrator Windows PowerShell window with two examples. Example 1 shows the command PS C:\>Get-Process and its output, which is a note about getting a list of all running processes. Example 2 shows the command PS C:\>Get-Process winword, explorer | format-list \*.

```

Administrator: Windows PowerShell
----- EXAMPLE 1 -----
PS C:\>Get-Process

This command gets a list of all of the running processes running on the
local computer. For a definition of each column, see the "Additional
Notes" section of the Help topic for Get-Help.

----- EXAMPLE 2 -----
PS C:\>Get-Process winword, explorer | format-list *

```

Figure 4-7

We don't want to set your expectations too high with the examples. While the core PowerShell cmdlets are very well documented and contain numerous examples (**Get-Service** has 11, if you can believe that!) cmdlets from other product teams may not be as robust. Often, the Help files for cmdlets that come from additional modules may only have a single example. It's a matter of how much time the product teams could invest in writing documentation. It will improve over time, but fortunately we will show you how to figure out how to read a cmdlet's syntax so that you can try your own solutions and only need to rely on examples for the time being.

Here is another way to see just the examples of a cmdlet.

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

The most complete Help for a cmdlet comes from using the **-Full** switch parameter. You will start using this switch almost exclusively when you run **Get-Help**. For now, **-Detailed** or **-Examples** is good enough, until we need the additional information that **-Full** provides.

## Another Help system

In addition to the cmdlet **Get-Help**, PowerShell includes a number of topic-oriented Help files. The **About\_Help** files contain information about scripting in PowerShell, along with other topics that will assist you in turning on or using special features.

Most admins forget about this Help system, but have no fear—when you use **Get-Help** to discover information (**Get-Help \*Array\***), both Help systems are searched. In other words, both the cmdlet-based Help and the **About\_**-based Help will be searched. This makes sure you don't miss anything important. Here's how you can find a full list of the Help files located in **About\_**.

```
Get-Help about_*
```

Name	Category	Module	Synopsis
about_Aliases	HelpFile		Descri...
about_Arithmetic_Operators	HelpFile		Descri...
about_Arrays	HelpFile		SHORT ...
about_Assignment_Operators	HelpFile		Descri...
about_Automatic_Variables	HelpFile		Descri...
about_Break	HelpFile		Descri...
about_Command_Precedence	HelpFile		Descri...
about_Command_Syntax	HelpFile		Descri...
about_Comment_Based_Help	HelpFile		SHORT ...
about_CommonParameters	HelpFile		SHORT ...
about_Comparison_Operators	HelpFile		SHORT ...
about_Continue	HelpFile		Descri...
about_Core_Commands	HelpFile		Lists ...
about_Data_Sections	HelpFile		Explai...
about_Debuggers	HelpFile		Descri...
about_Do	HelpFile		Runs a...
about_Environment_Variables	HelpFile		Descri...

Figure 4-8

If you find a topic that you want to know more about, just type **Get-Help** (remember you can use **Help** or **Man** as well) to see the entire file.

```
Get-Help about_Aliases
```

Administrator: Windows PowerShell	
PS C:\> Get-Help about_Aliases	
<b>TOPIC</b>	
about_aliases	
<b>SHORT DESCRIPTION</b>	
Describes how to use alternate names for cmdlets and commands in Windows PowerShell.	
<b>LONG DESCRIPTION</b>	
An alias is an alternate name or nickname for a cmdlet or for a command element, such as a function, script, file, or executable file. You can use the alias instead of the command name in any Windows PowerShell commands.	
To create an alias, use the New-Alias cmdlet. For example, the following command creates the "gas" alias for the Get-AuthenticodeSignature cmdlet:	
New-Alias -Name gas -Value Get-AuthenticodeSignature	
After you create the alias for the cmdlet name, you can use the alias instead of the cmdlet name. For example, to get the Authenticode signature	

Figure 4-9

**Tip:**

Tab completion will help complete the `About_` name you are typing. This is good to keep in mind as some of the Help files have long names.

Of course, sometimes it can be a distraction to have to refer to Help while you're trying to work out a command line and sometimes you don't want to have to page through the information the way the Help function does. If you prefer an on-screen, electronic cmdlet reference, there are several options available (including **-Online** or **-ShowWindow**). We also suggest that you go to <http://www.sapien.com> and download the free PowerShell Help tool or to the Apple App Store and buy iPowerShell Pro. They provide a nicely formatted version of Help in a graphical window, which you can have up and running alongside your PowerShell console or on your iPhone or iPad, giving you access to the built-in Help without distracting you from the command you're trying to construct.

With all this help, now it's time to break apart how a cmdlet is used in the console.

## Cmdlet anatomy

At first, cmdlet syntax can seem confusing, with many strange symbols representing different things. Being able to read the syntax of a cmdlet is one of the most important actions to be able to use a cmdlet correctly and to its fullest.

## How cmdlets work

PowerShell commands begin with a cmdlet name, followed by parameters and arguments that control the output of the command. As you saw earlier with **Get-Service**, some cmdlets produce output without any parameters; but parameters allow you to control the cmdlet if you don't want the default results.



Figure 4-10

Parameters always begin with a dash—in fact a common mistake is to forget to include this. Some parameters require additional information or arguments. Let's say I want only the **bits** and **bfe** services. Here is an example of using the **-Name** parameter and supplying the arguments **bits** and **bfe**.

```
Get-Service -name bits, bfe
```

```
Administrator: Windows PowerShell
PS C:\> Get-Service -name bits, bfe
Status    Name          DisplayName
-----   --
Running   bfe          Base Filtering Engine
Stopped   bits         Background Intelligent Transfer Ser...
PS C:\>
```

Figure 4-11

Notice that the parameter **-Name** allowed us to control the output of the **Get-Service** cmdlet. Many arguments can contain wildcard characters, so if we wanted to see all services that begin with 'B' and are from a remote computer called 'DC', then we would use the following:

```
Get-Service -Name b* -ComputerName DC
```

```
Administrator: Windows PowerShell
PS C:\> Get-Service -Name b* -ComputerName DC
Status    Name          DisplayName
-----   --
Running   BFE          Base Filtering Engine
Stopped   BITS         Background Intelligent Transfer Ser...
Running   BrokerInfrastr... Background Tasks Infrastructure Ser...
Stopped   Browser       Computer Browser
PS C:\>
```

Figure 4-12

Like command-line utilities you may have used in the past, PowerShell cmdlets often support a number of parameters. However, unlike the old command-line utilities, PowerShell's cmdlet parameters use a consistent naming pattern, which makes the parameters easier to learn. For example, the **Get-Content** and **Set-Content** cmdlets allow you to specify a path—such as a file path—and so both use the same parameter name, **-Path**, for that parameter.

PowerShell uses spaces as parameter delimiters.

```
Get-Content -Path C:\Content.txt
```

If a parameter value contains spaces, then you must enclose the value in either single or double quotation marks.

```
Get-Content -Path "C:\Test Files\content.txt"
```

## Positional parameters

Typically, the most commonly used parameter for any given cmdlet is positional, meaning you don't even have to specify the parameter name. Therefore, the example in Figure 4-13 is also valid.

```
Get-Service bits
```

The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command 'PS C:\> Get-Service bits' is entered. The output is a table:

Status	Name	DisplayName
Stopped	bits	Background Intelligent Transfer Ser...

PS C:\> \_

Figure 4-13

The PowerShell interpreter is pretty smart. If only a series of arguments are placed after the command, the parameter binder in the interpreter figures out which argument goes with which parameter. It does this by following the described order of the parameters, which can be found in the Help system.

This does bring up another common mistake—especially when first learning PowerShell—the parameter/argument/dash typing mistake.

```
Get-Service -bits
```

Notice the mistake? When you don't use parameters, sometimes you might type a dash in front of the argument. This will not work, so its best practice—especially at first—to always use the parameter names. Also, as you will see, if you always use the parameter names, you can find Help on the parameters.

## Shortening parameter names

What's more, when you do need to type a parameter name, you need to type only as much of the name as necessary to distinguish the parameter from others. For example, here's a command that will retrieve process information from a remote computer called DC.

```
ps -comp dc
```

Administrator: Windows PowerShell							
Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
351	32	14484	16368	1150		1624	certsrv
267	11	1476	3900	47		540	csrss
132	13	1332	7760	48		608	csrss
320	30	14744	18044	623		1672	dfssrs
119	11	1908	5280	33		1952	dfssvc
189	14	3488	10228	55		2584	dllhost
5221	3613	47128	48224	96		1732	dns
178	14	18868	27984	106		432	dwm
797	42	13312	41084	236		952	explorer
0	0	0	20	0		0	Idle
138	14	4696	11644	56		1768	inetinfo
92	11	1652	4296	30		1804	ismserv

Figure 4-14

Notice the **-ComputerName** can be typed as **-comp** because no other parameter has those four characters. Pretty smart of PowerShell to do this and \*nix folks love it. For most Windows admins, this makes the command more confusing and, for the sake of clarity, we generally don't recommend it (after all, you can use tab completion). However, you will find commands like this on the Internet, so you should know that it's possible.

### Wait a minute!

So, you might be asking how did we know that the cmdlets in the above example had these parameters? How did we know what kind of arguments to give them? Well, one of the best parts of the Help files on cmdlets is reading the syntax. It spells it all out for you.

## Getting into syntax

When you run Help on a cmdlet, the third topic listed is the syntax of the cmdlet. This includes the names of the parameters it supports, along with the types of arguments that a parameter might accept.

### Parameter sets

First and most confusing to admins is that a cmdlet may have more than one syntax. These are referred to as parameter sets. If a cmdlet has more than one syntax, it is because there is a parameter that is unique to that syntax. In the example below, notice that **Stop-Service** has three different syntaxes. Each

syntax has unique parameters that will cause the cmdlet to behave differently. The definitions of the parameters further down the Help file will help explain this, but we will get to that shortly.

```
Get-Help Stop-Service
```

```
Administrator: Windows PowerShell
PS C:\> Get-Help Stop-Service

NAME
  Stop-Service

SYNOPSIS
  Stops one or more running services.

SYNTAX
  1 Stop-Service [-InputObject] [-Exclude] [-Force] [-Include] [-PassThru]
    [-Confirm] [-WhatIf] [<CommonParameters>]

  2 stop-Service [-Exclude] [-Force] [-Include] [-PassThru] -DisplayName
    [-Confirm] [-WhatIf] [<CommonParameters>]

  3 stop-Service [-Name] [-Exclude] [-Force] [-Include] [-PassThru] [-Confirm]
    [-WhatIf] [<CommonParameters>]
```

Figure 4-15

Get used to the fact that cmdlets can have several parameter sets—it truly increases flexibility, as you will see throughout this book. Something else not to be startled by is the number of parameters. Later in this book you will see cmdlets that could have hundreds of parameters.

## Syntax

Being able to read the symbols surrounding the parameters and arguments will describe what is required and not when using the cmdlet.

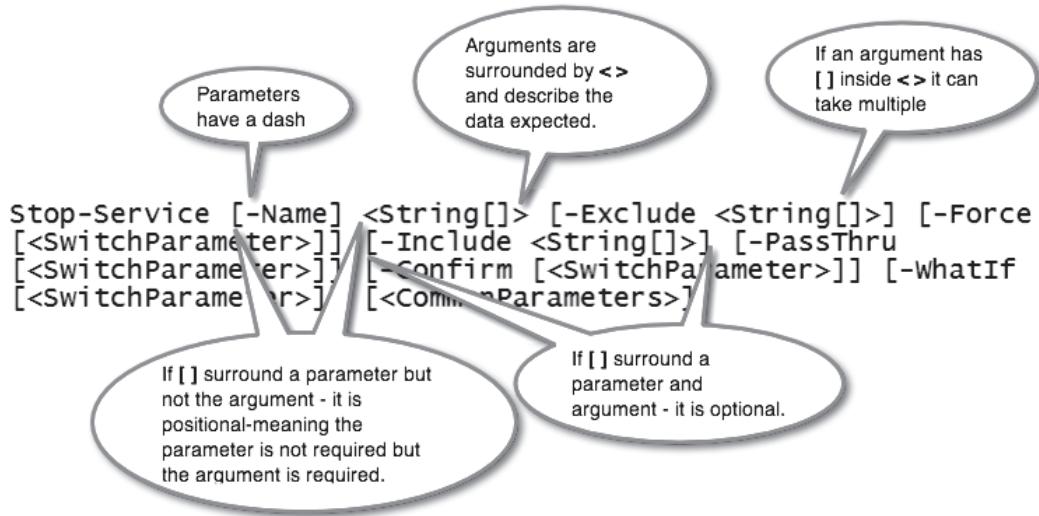


Figure 4-16

**Note:**

While the syntax is capitalized for readability, Windows PowerShell is case-insensitive.

Parameters appear in order. The order of parameters is significant only when the parameter names are positional. If you do not specify parameter names when you use a cmdlet, Windows PowerShell assigns values to parameters by position and by type.

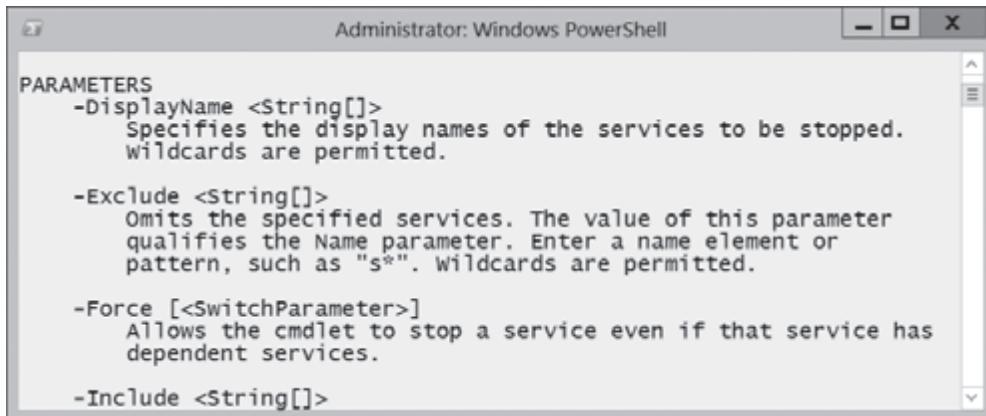
Parameter names are preceded by a hyphen (-). Switch parameters appear without a value type.

A parameter name can be optional even if the value for that parameter is required. This is indicated in the syntax by the brackets around the parameter name, but not the parameter type.

### **Stop! How do I know what all these parameters mean?**

This is the best part. When you are examining the syntax for a cmdlet, learning what is and is not required, seeing what arguments a cmdlet will accept, you will often see parameter names that don't make sense. How do you figure it out? You don't need to—it's all written into the Help file. Just use **-Detailed** or **-Full** to find out more information.

```
Get-Help Get-Service -Detailed
```



The screenshot shows the Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "Get-Service". The output displays detailed help information for the cmdlet, including parameters such as -DisplayName, -Exclude, -Force, and -Include, along with their descriptions.

```

PARAMETERS
  -DisplayName <String[]>
    Specifies the display names of the services to be stopped.
    Wildcards are permitted.

  -Exclude <String[]>
    Omits the specified services. The value of this parameter
    qualifies the Name parameter. Enter a name element or
    pattern, such as "s*". Wildcards are permitted.

  -Force [<SwitchParameter>]
    Allows the cmdlet to stop a service even if that service has
    dependent services.

  -Include <String[]>

```

Figure 4-17

And this brings us to the moral of the story and back to where we started. You have everything you need to find cmdlets, figure out how they work and what the parameters will do, and even see examples. As you add more modules (and even more cmdlets) the Help files will explain everything to you. The important part is this: if you are not willing to use these Help files, you won't be able to use PowerShell. To help you with this, not only is there an exercise at the end of this chapter to give you some experience, we will review this often throughout the book.

## SAPIEN Help

Both SAPIEN's PrimalScript and PowerShell Studio provide help—in many forms. The automatic IntelliSense (PrimalSense) will display the syntax and parameters for a cmdlet as you type. Yet both also support dynamic help that will appear in the Help screen if you have it turned on. Figure 4-18 is an example of PrimalScript.

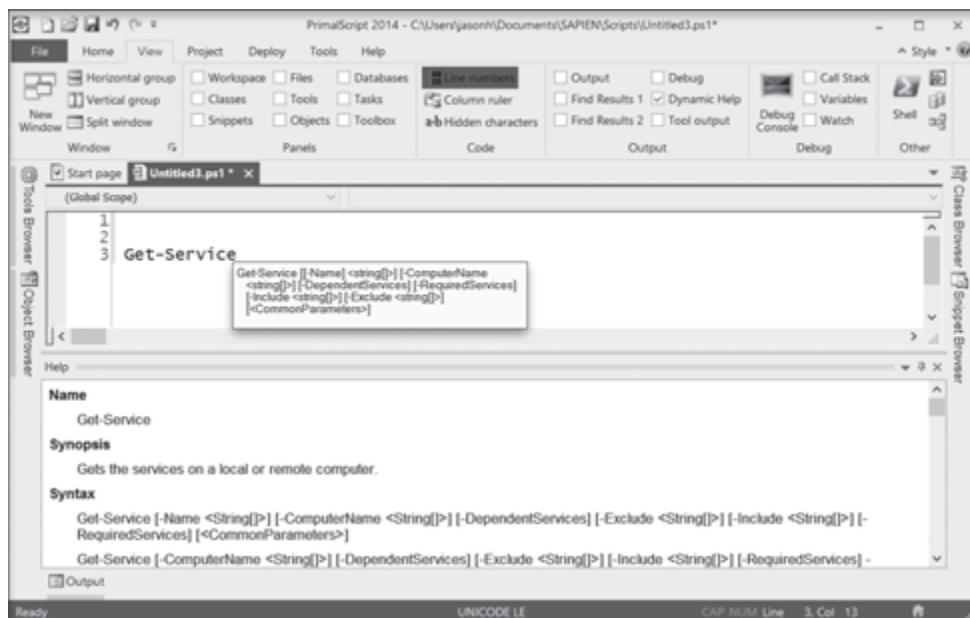


Figure 4-18

## Finding Help when needed

Figure 4-19 shows this in PowerShell Studio.

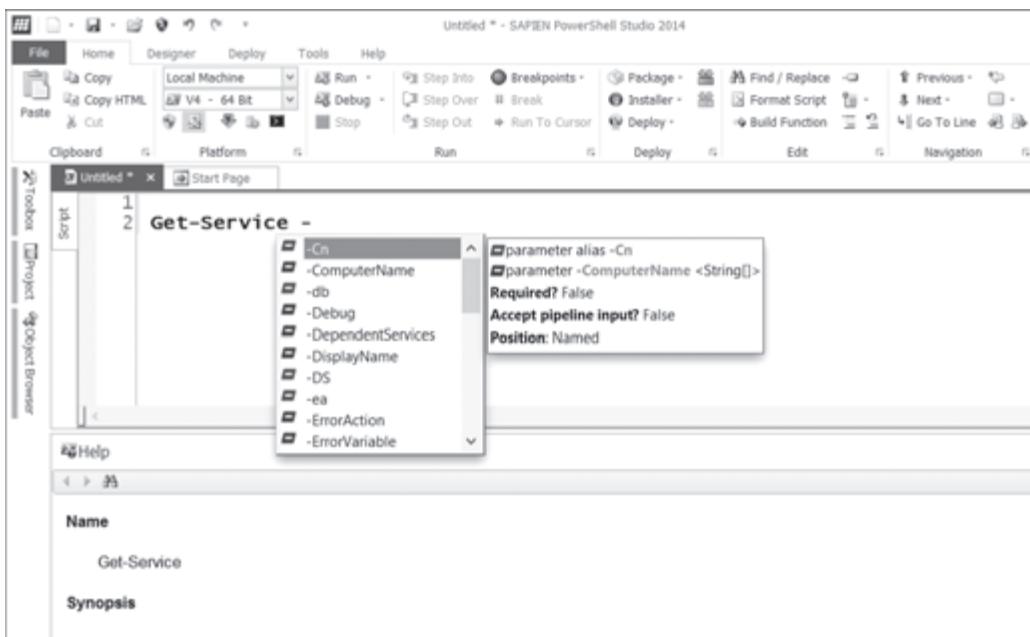


Figure 4-19

Notice how the Help file is displayed in the lower pane. You can also select a cmdlet and press F1 to view the full Help file.

With everything that has been discussed in this chapter, it's time to try it out for yourself.

## Exercise 4 – Finding Help when needed

Time: 30 minutes

The goal in this exercise is to discover common cmdlets, and start to open up and read the Help files. Treat each of these as a little mystery to solve. Don't forget that the Help system supports the \* wildcard (**Get-Help \*service\***)—so start by trying to think of a useful noun to look for. Just as with Bing or Google, sometimes you will need to narrow or expand your search.

### Task 1

Are there any cmdlets that can redirect output to a printer?

### Task 2

Can you find any cmdlets capable of converting the output of another cmdlet into XML?

### Task 3

How many cmdlets are available for working with processes?

### Task 4

How can you retrieve a list of installed services or processes from a remote computer?

### Task 5

What cmdlet would you use to write to an event log?

### Task 6

You want to retrieve the last 10 entries from the system event log, but you only want the errors. Can you find the parameters that will help you narrow the search? Try it if you like!

### Task 7

By default, **Out-File** overwrites existing files with the same name. Is there a way to prevent that?

## Task 8

How could you find a list of all aliases in PowerShell?

## Task 9

What cmdlets are available to create, modify, export, or import aliases?

## Task 10

PowerShell contains something called Switch. Can you locate a Help topic on this? Can you find an example of how to use this?

Note: Don't actually try to run this, as we haven't discussed it yet.



## Chapter 5

# Working with providers and drives

## PowerShell PSProviders and PSDrives

PowerShell introduces a unique concept called PSDrives, or PowerShell Drives. There's an interesting philosophy behind these: the team that created PowerShell knew that they would have an uphill battle convincing admins to drop their graphical tools and turn to the command line. They figured the switch would be easier if they could leverage the relatively small set of command-line skills that most Windows admins already had. PowerShell's cmdlets, with their command-line parameters, are one example of that. Most admins are already familiar with command-line utilities and switches and PowerShell simply expands on that familiarity, adding in better consistency, for a shorter learning curve. The other main skill that the team wanted to leverage was the ability to navigate a complex hierarchical object store. Bet you didn't know you had that skill, but you do!

PowerShell provides three cmdlets for working with PSDrives:

- **Get-PSDrive** retrieves information about a specific PSDrive or, if you don't specify one, lists all available drives.
- **New-PSDrive** creates a new drive using the specified provider. Use this, for example, to map network drives.
- **Remove-PSDrive** removes an existing drive from the shell.

Here's a quick example of the drive you probably didn't even know you had access to.

`Get-PSDrive`

Name	Used (GB)	Free (GB)	Provider	Root
A			FileSystem	A:\
Alias			Alias	
C	10.75	48.90	FileSystem	C:\
Cert			Certificate	\
D			FileSystem	D:\
Env			Environment	
Function			Function	
HKCU			Registry	HKEY_CURRENT_...
HKLM			Registry	HKEY_LOCAL_MA...
Variable			Variable	
WSMan			WSMan	
Z	450.03	248.85	FileSystem	\\vmware-host...

Figure 5-1

Windows has several different hierarchical stores, aside from the file system. The registry for example, looks a lot like the file system, don't you think? It has folders (registry keys) and files (registry settings), and the files have contents (the values within settings). The Certificate Store in Windows is similar, too. So is Active Directory, for that matter.

PowerShell lets you leverage all of these hierarchical stores by using the same techniques that you use to work with the file system. Open a PowerShell console and run **Get-PSDrive**. You'll see a list of all the drives attached to your PowerShell console and you'll see the provider that connects each drive. For example, you'll doubtless see drives C: and D:, and perhaps others, using the **FileSystem** provider—and these drives are the ones you're probably already familiar with. However, you'll also see drives HKCU: and HKLM:, which use the **Registry** provider. You'll see a CERT: drive for the Certificate Store, and an ENV: drive for environment variables. Other drives like Function: and Variable: connect to PowerShell's own internal storage mechanisms.

Try accessing the HKEY\_LOCAL\_MACHINE hive of the registry. How? The same way you'd access a familiar drive letter like the D: drive.

```
cd hklm:
```

Simply change to the HKLM: drive and you're there. Need to see the keys that are available at list level? Ask for a list.

```
dir
```

Want to change into the software key? You can probably guess how that's done.

```
cd software
```

```

Administrator: Windows PowerShell
PS C:\> cd HKLM:\SOFTWARE
PS HKLM:\SOFTWARE> dir

Hive: HKEY_LOCAL_MACHINE\SOFTWARE

Name          Property
----          -----
Classes
Clients
Microsoft
ODBC
Policies
RegisteredApplications      Paint : SOFTWARE\Microsoft\Windows\CurrentVersion\App1

```

Figure 5-2

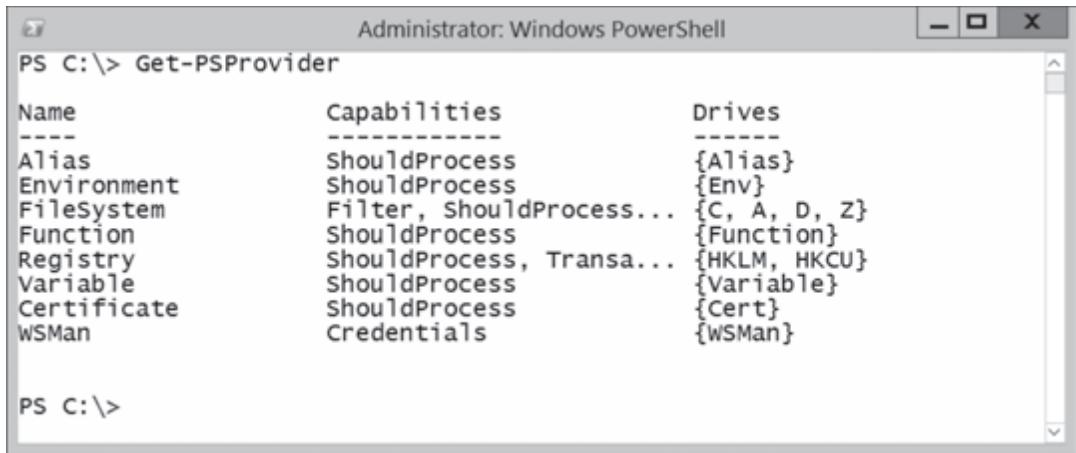
Note that PowerShell isn't even case sensitive! Want to delete a registry key (be careful)? The **Del** command will do it. There is much more that you can do in the registry and that is covered in the reference section "In Depth 06—Managing the Registry."

Additional drives are added depending on the installed software or loaded module. As an example, there are drives for IIS (IIS:), Active Directory (AD:), and SQL Server (SQLSERVER:). **Get-PSDrive** is your window into seeing which drives are available.

## PSProviders

How does this all work? PowerShell uses PSProviders (provider), which makes different hierarchical storage appear as a disk drive. The provider is listed when you use the **Get-PSDrive** cmdlet; however, you can directly access the current providers on your system by using **Get-PSProvider**.

**Get-PSProvider**



```
Administrator: Windows PowerShell
PS C:\> Get-PSProvider

Name          Capabilities      Drives
----          -----
Alias         ShouldProcess    {Alias}
Environment   ShouldProcess    {Env}
FileSystem   Filter, ShouldProcess... {C, A, D, Z}
Function     ShouldProcess    {Function}
Registry     ShouldProcess, Transa... {HKLM, HKCU}
Variable     ShouldProcess    {Variable}
Certificate  ShouldProcess    {Cert}
WSMan        Credentials      {WSMan}

PS C:\>
```

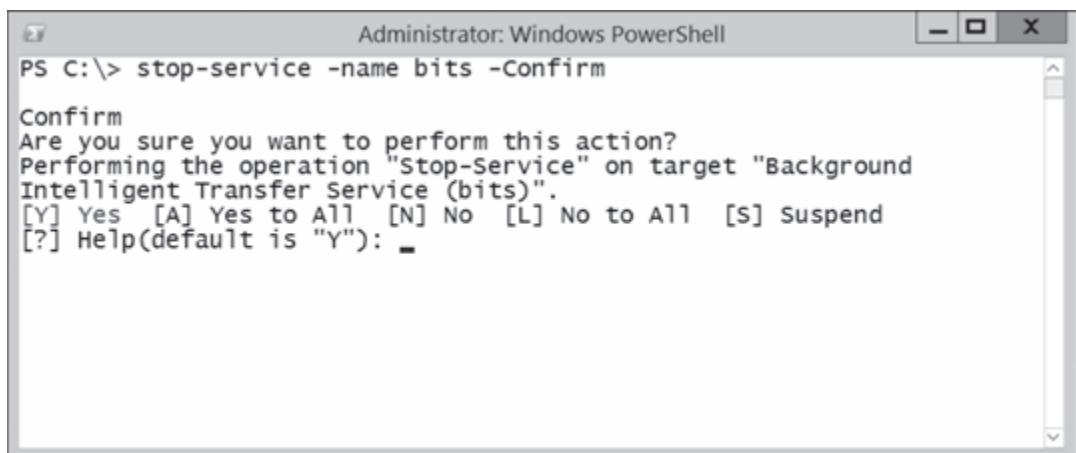
Figure 5-3

The display returns a list of the currently installed providers, the capabilities of the provider, and the drive that they expose, as shown in Figure 5-3. Microsoft has no idea what future products may need a provider, so this list is dynamic—in other words, when you load modules or install software—this list can change.

## Capabilities

The capabilities list indicates what the provider can do for you. In the case of **ShouldProcess**, this means that the provider supports the parameter switches **-WhatIf** and **-Confirm** (these are detailed later, but in short, they provide you a safety net). As an example, here's what you would do if you wanted to receive a confirmation message before stopping a service.

```
stop-service -name bits -Confirm
```



```
Administrator: Windows PowerShell
PS C:\> stop-service -name bits -Confirm

Confirm
Are you sure you want to perform this action?
Performing the operation "Stop-Service" on target "Background
Intelligent Transfer Service (bits)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend
[?] Help(default is "Y"): _
```

Figure 5-4

The **Filter** capability indicates that **-Filter** is available, permitting you to restrict your results list. Think of the DOS command that lets you look for all files in the current directory.

```
dir *.*
```

The new version of this would be to use **Get-ChildItem** on your C: drive (using the **FileSystem** provider). It supports the use of **-Filter**.

```
Get-ChildItem -Path c: -Filter *.*
```

Mode	LastWriteTime	Length	Name
d----	6/26/2013 5:28 PM		inetpub
d----	6/15/2013 10:47 PM		PerfLogs
d-r--	6/29/2013 10:30 AM		Program Files
d----	6/29/2013 10:38 AM		Program Files (x86)
d----	6/29/2013 10:43 AM		Repository
d----	6/29/2013 1:07 PM		scripts
d-r--	6/26/2013 5:10 PM		Users
d----	6/29/2013 9:51 AM		Windows

Figure 5-5

Besides **Filtering** and **ShouldProcess**, some providers support using alternative credentials, such as the WSMAN provider. Using the drive associated with the provider permits you to use the **-Credential** parameter to specify an alternate credential.

## Getting Help for the provider

In the last chapter, you learned about using **Get-Help** (or **Help/Man**) to find information about cmdlets and the **About\_\*** Help system. Providers also offer help in the same format as cmdlets—description, examples—basically how the provider works and what cmdlets you can use with it. Here’s an example of accessing Help for the **FileSystem** provider.

```
Get-Help FileSystem
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-Help FileSystem" is run at the prompt. The output provides details about the FileSystem provider, including its provider name (FileSystem), drives (C, D), synopsis (Provides access to files and directories), and description (The Windows PowerShell FileSystem provider lets you get, add, change, clear, and delete files and directories in Windows PowerShell).

```

Administrator: Windows PowerShell
PS C:\> Get-Help FileSystem

PROVIDER NAME
    FileSystem

DRIVES
    C, D

SYNOPSIS
    Provides access to files and directories.

DESCRIPTION
    The Windows PowerShell FileSystem provider lets you get, add,
    change, clear, and delete files and directories in Windows
    PowerShell.
  
```

Figure 5-6

We find it useful to examine the provider Help anytime a new one is added to our system, but keep in mind that not all the other product teams that make providers for their products include Help.

## Managing drives and providers

It's time to start learning how to use providers and drives to your advantage with PowerShell. There are cmdlets (with common aliases) that will make the job of working with the registry and other drives easier. Remember how this chapter started out—**PSDrives** were created to help bridge the gap and create a comfortable environment for admins managing products. As the product teams have added more practical cmdlets, the need to manage through a **PSDrive** and **PSProvider** is diminishing. In fact, we find it more challenging to manage using PSDrives and prefer using product specific cmdlets. Today, with PowerShell version 4 and the thousands of cmdlets in Windows 8.x and Server 2012 (R2), you will find that cmdlets make your life easier, but there will always be a time, such as when a product doesn't have full cmdlet support—and knowing how to work with PSDrives is a benefit.

Let's get started!

## Mapping drives

You can create your own drives by using the providers that you have installed. For example, here's how to map your Z: drive to \\Server\Scripts.

```
New-PSDrive -Name Z -PSPrinter FileSystem -Root \\Server\Scripts
```

```

Administrator: Windows PowerShell
PS C:\> New-PSDrive -Name Z -PSProvider FileSystem -Root \\Server\Scripts
pts
Name          Used (GB)    Free (GB) Provider      Root
----          -----       -----      -----
Z                         Filesystem   \\Server\Scripts

PS C:\> cd z:
PS Z:\> dir

Directory: \\Server\Scripts

Mode          LastWriteTime      Length Name

```

Figure 5-7

The **-PSProvider** parameter tells PowerShell exactly which provider you're using. You can even map to local folders.

```
New-PSDrive -Name Z -PSProvider FileSystem -root C:\test
```

This maps the Z: drive to the local C:\Test folder. Unfortunately, PowerShell doesn't provide any means for using the other providers remotely. Mapping to a remote UNC is about your only option and that only works with the **FileSystem** provider. You can't map to remote registries or certificate stores. That'd be a useful capability, but it doesn't exist.

You can remove a mapped drive by using **Remove-PSDrive**.

```
Remove-PSDrive -Name Z
```

Any mappings that you create in PowerShell are preserved only for the current session. Once you close PowerShell, they're gone—unless you add the **-Persist** parameter. Also, your drive mappings don't show up in Windows Explorer—they only exist in PowerShell, unless you again use the **-Persist** parameter. If you need to re-create a particular mapping each time you open a PowerShell console, then add the appropriate **New-PSDrive** command to a PowerShell profile (we'll talk more about this later, when we get to scripting).

You should pay special attention to the fact that PowerShell's drives exist only within PowerShell itself. For example, if you map the Z: drive to a UNC, and then try to launch Windows Notepad to open a file on the Z: drive, it won't work. That's because the path is passed to Notepad, which has to ask Windows, not PowerShell, to get the file. Since Windows doesn't "have" the Z: drive, the operation will fail.

## Mapping drives other ways

So, you're probably wondering if there is a better way to map drives than using the **New-PSDrive** cmdlet. If it is a remote computer and you want to map to a share (not use one of the other providers), then there is always the Windows command **Net Use**. This is not a PowerShell command; but remember, PowerShell lets you run commands native to Windows.

```
Net use F: \\Server\Scripts
```

```

Administrator: Windows PowerShell
PS C:\> Net use F: \\server\scripts
The command completed successfully.

PS C:\> f:
PS F:\> dir

Directory: F:\

Mode          LastWriteTime    Length Name
----          -----          ---- 
-a---  6/29/2013 1:05 PM      797 remotetest.ps1

PS F:\> notepad

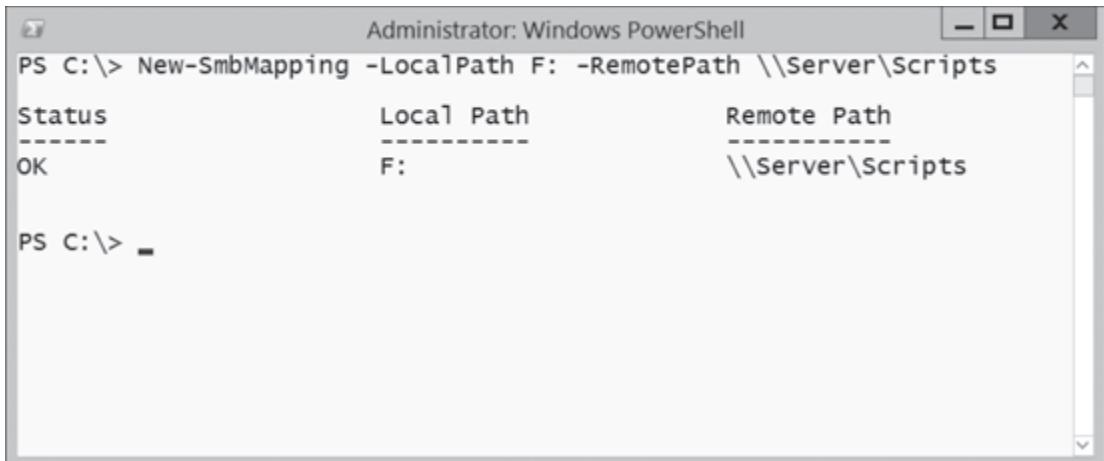
```

Figure 5-8

The advantage to using the Windows command **Net Use** is that this does create the mapping in Windows. So by launching Notepad, you could open a file on the remote computer by using the drive letter F:. We know what you're thinking: then why bother with **New-PSDrive**? Well, **New-PSDrive** can work with other providers, the **Net Use** command is only for mapping to UNC's.

Starting with PowerShell version 3 and later on Windows 8 and Server 2012, a new set of Server Message Block (SMB) cmdlets have been added. This lets you both share and map drives using the new SMB 3.0 features. Use **Get-Help** to get a list of SMB cmdlets, but use **New-SMBMapping** to map a drive to Server 2012.

```
New-SmbMapping -LocalPath F: -RemotePath \\Server\Scripts
```



```
Administrator: Windows PowerShell
PS C:\> New-SmbMapping -LocalPath F: -RemotePath \\Server\Scripts
Status          Local Path          Remote Path
-----          -----
OK              F:                \\Server\Scripts

PS C:\> -
```

Figure 5-9

There are many advantages to SMB 3.0, but also to the cmdlets. In PowerShell versions 1 and 2, you needed to use Windows native commands to manage shares and drive mappings. Now, in PowerShell version 4, you have cmdlets that allow you to set all of the features of SMB 3.0, including permissions and persistence.

## **Listing child items**

When you have a file system drive (or other provider) that you want to work with, you will want to find information about the contents of that drive. This is where we take a deeper look at **Get-ChildItem** (and its alias, **Dir**).

PowerShell thinks of everything as an object—a concept you will become more familiar with throughout the next few chapters. A folder on your hard drive, for example, is an object. Of course, folders have subfolders and files, which PowerShell thinks of as children of the folder. That's not an uncommon term. For example, many of us are already accustomed to thinking of parent folders and so forth. So, if you're working with a particular folder, meaning that PowerShell is inside that folder, then the way you'd get a list of child items is simple: **Get-ChildItem**.

Remember, PowerShell cmdlets always use a singular noun, so it's not **Get-Children** or **Get-ChildItems**—it's **Get-ChildItem**. Typed alone, the cmdlet—or one of its aliases, such as **Dir**, **Ls**, or **GCI**—will return a list of child items for the current object; that is, the folder the shell is currently in.

**Get-Childitem**

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "PS C:\> Get-Childitem" is run, followed by the output:

```

Directory: C:\

Mode                LastWriteTime      Length Name
----                -----          ---- 
d----
```

Mode	LastWriteTime	Length	Name
d----	6/26/2013 5:28 PM		inetpub
d----	6/15/2013 10:47 PM		PerfLogs
d-r--	6/29/2013 10:30 AM		Program Files
d----	6/29/2013 10:38 AM		Program Files (x86)
d----	6/29/2013 10:43 AM		Repository
d----	6/29/2013 1:07 PM		scripts
d-r--	6/26/2013 5:10 PM		Users
d----	6/29/2013 9:51 AM		Windows

Figure 5-10

By default, this information is displayed in a table format, as shown in Figure 5-10. However, if you ask for Help on **Get-ChildItem**, you'll see that it has a lot of additional options, which are exposed via parameters. For example, one useful parameter is **-Recurse**, which forces the cmdlet to retrieve all child items, even those deeply nested within subfolders—similar to the old DOS **Dir** command switch **/S**.

You can find the child items for a specific path, too.

```
gci -Path c:\test
```

Notice that we've used the **GCI** alias, and specified the name of the **-Path** parameter. The online Help indicates that the actual parameter name is optional, in this case, because the first parameter is positional. Therefore, the following example would achieve the same thing.

```
dir c:\test
```

Of course, we used a different alias, but it doesn't matter. The command works the same either way. Other parameters let you filter the results. For example, consider the following:

```
Dir c:\scripts -Exclude *.ps1
```

```
Administrator: Windows PowerShell
PS C:\> Dir c:\scripts -Exclude *.ps1

Directory: C:\scripts

Mode                LastWriteTime      Length Name
----                -----          ---- 
-a---       6/30/2013 12:55 PM        21 HelpFullNotes.txt
-a---       6/30/2013 12:27 PM     197 scripts.recall

PS C:\> -
```

Figure 5-11

The **-Exclude** parameter accepts wildcards, such as \* and ?, and removes matching items from the result set. Similarly, the **-Include** parameter filters out everything except those items that match your criteria. One important thing to remember about **-Include** and **-Exclude** is that they force the cmdlet to retrieve all of the child items first, and then filter out the items you didn't want. That can sometimes be slow, when a lot of items are involved. Read that again! An alternate technique is to use the **-Filter** parameter. Its use differs depending on the **PSDrive** provider you're working with, although with the file system it uses the familiar \* and ? wildcards.

```
Dir c:\scripts -filter *.ps1
```

```
Administrator: Windows PowerShell
PS C:\> Dir c:\scripts -filter *.ps1

Directory: C:\scripts

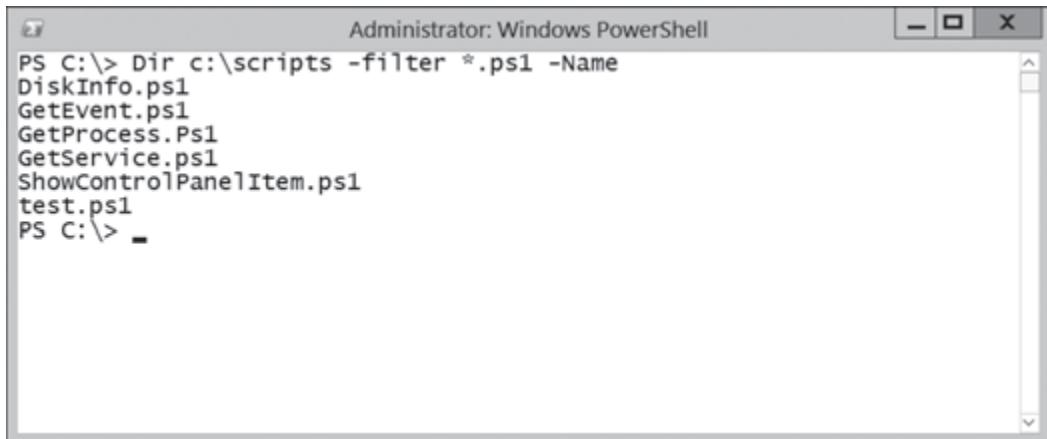
Mode                LastWriteTime      Length Name
----                -----          ---- 
-a---       6/29/2013 1:09 PM        869 DiskInfo.ps1
-a---       5/3/2013  6:31 PM         57 GetEvent.ps1
-a---       5/3/2013  6:46 PM         91 GetProcess.ps1
-a---       5/3/2013  6:36 PM         49 GetService.ps1
-a---      2/25/2012 2:12 PM       580 ShowControlPanelItem.ps1
-a---      6/28/2013 8:47 AM          11 test.ps1
```

Figure 5-12

## Windows PowerShell: TFM

Only items matching your criteria are included in the output. If that output contains too much information, you can just have the cmdlet return the names of the child items.

```
Dir c:\scripts -filter *.ps1 -Name
```



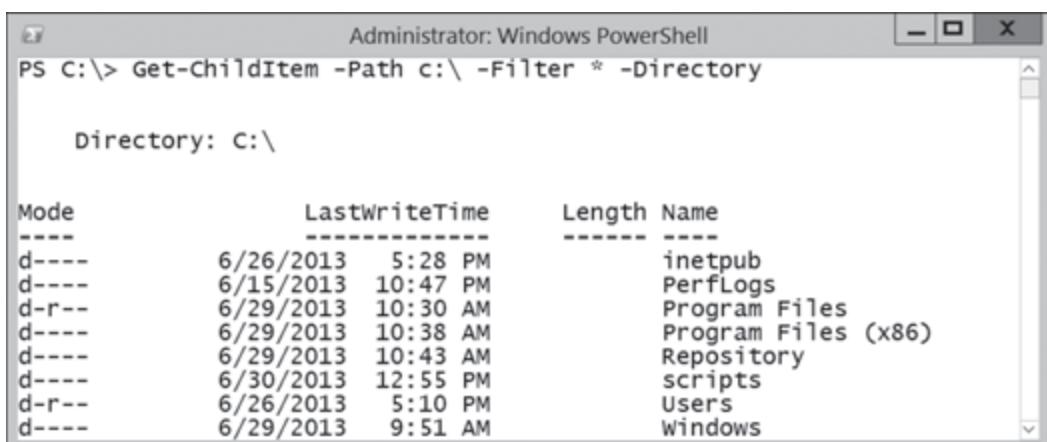
```
Administrator: Windows PowerShell
PS C:\> Dir c:\scripts -filter *.ps1 -Name
DiskInfo.ps1
GetEvent.ps1
GetProcess.Ps1
GetService.ps1
ShowControlPanelItem.ps1
test.ps1
PS C:\> _
```

Figure 5-13

In Figure 5-13, by combining the **-Filter** and **-Name** parameters, we've generated a very customized list: just the names of the PowerShell scripts in this folder.

Often, you will receive a list of both files and folders. In PowerShell versions 3 and 4, **Get-ChildItem** supports the parameter switches **-File** and **-Directory**. Here's what you would do if you want to see all the folders in your root drive, without any files.

```
Get-ChildItem -Path c:\ -Filter * -Directory
```



```
Administrator: Windows PowerShell
PS C:\> Get-ChildItem -Path c:\ -Filter * -Directory

Directory: C:\

Mode                LastWriteTime      Length Name
----                -----          ---- 
d----
```

Mode	LastWriteTime	Length	Name
d----	6/26/2013 5:28 PM		inetpub
d----	6/15/2013 10:47 PM		PerfLogs
d-r--	6/29/2013 10:30 AM		Program Files
d----	6/29/2013 10:38 AM		Program Files (x86)
d----	6/29/2013 10:43 AM		Repository
d----	6/30/2013 12:55 PM		scripts
d-r--	6/26/2013 5:10 PM		Users
d----	6/29/2013 9:51 AM		Windows

Figure 5-14

Occasionally, PowerShell can annoy you by attempting to interpret characters in a path as a wildcard. For example, in the file system, the question mark character is used as a single character wildcard.

```
dir t?st.txt
```

However, within the Windows registry, the question mark character is a legitimate character. To see how this works temporarily, create some new keys in the registry.

```
cd HKCU:\Software\
mkdir Micr?soft
mkdir Micr?soft\test1\
```

Now, try the following:

```
dir Micr?soft -recurse
```

You might be expecting a listing of registry keys underneath the key “Micr?Soft,” but PowerShell interprets the question mark as a wildcard, and will instead search for any key like “MicrzSoft,” “Micr0soft,” and so forth. If you run into this situation, just use a slightly different technique.

```
dir -literalPath Micr?soft -recurse
```

Here, the **-literalPath** parameter tells PowerShell to take the path literally—that is, to not try and interpret any characters as wildcards. Now you should see only the specified key and its children.

Finally, remember that PowerShell is designed to work with a variety of different storage systems. When you’re working with the CERT: drive—the disk drive that’s connected to your local certificate store—**Get-ChildItem** supports a parameter named **-codeSigningCert**. It filters the display of child items to those that are code-signing certificates, rather than other types. This makes it easier to retrieve a code-signing certificate when you want to digitally sign a PowerShell script file.

```
cd cert:
Get-ChildItem -CodeSign
```

Notice that we didn’t specify the full name of **-codeSigningCert**; we didn’t need to, because you only need a few characters to differentiate the parameter name from the other ones available (actually, we could have used fewer characters, but this way it’s still relatively obvious what’s going on when you read the command line).

## Changing location

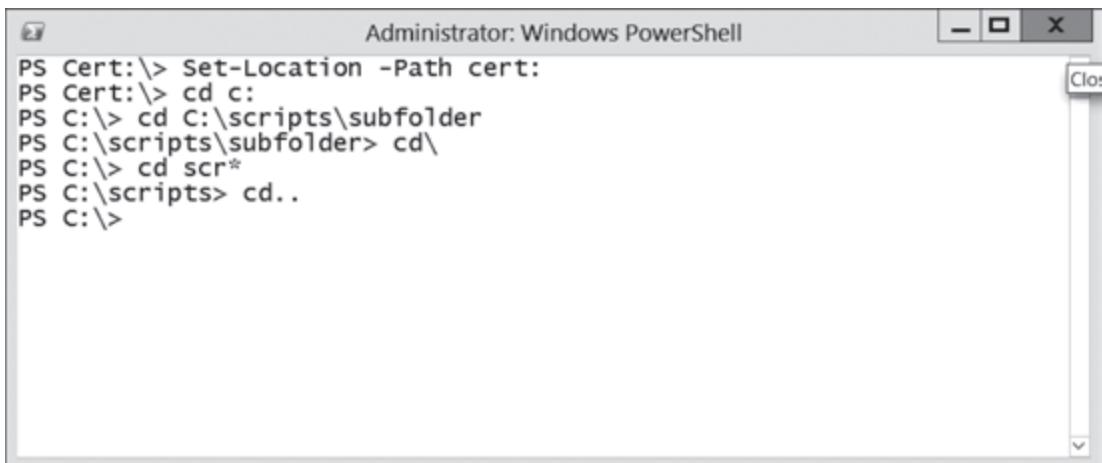
Now that you know how to get a list of child items from a single location, you'll also need to know how to change locations. In MS-DOS and \*nix, that's done with the **CD** command (short for Change Directory), and in many operating systems the longer **ChDir** command will also work. PowerShell aliases **CD** to **Set-Location**.

Generally speaking, you just tell **Set-Location** where you want to go.

```
Set-Location -Path CERT:
```

Or, use an alias and omit the parameter name.

```
cd C:\scripts\subfolder
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history pane displays the following sequence of commands:

```

PS Cert:\> Set-Location -Path cert:
PS Cert:\> cd c:
PS C:\> cd C:\scripts\subfolder
PS C:\scripts\subfolder> cd\
PS C:\> cd scr*
PS C:\scripts> cd..
PS C:\>

```

Figure 5-15

As with the **Get-ChildItem** cmdlet, you can also specify a literal path, if you don't want PowerShell interpreting wildcard characters.

```
cd -literal HKCU:\SOFTWARE\Manu?\Key
```

If you're curious, you can provide wildcards if you don't use the **-literalPath** parameter.

```
cd scri*
```

On the test system, the above changes into the C:\Scripts folder. You'll see an error if the path you specify resolves to more than one path. This is different than the Cmd.exe behavior of simply changing into the first matching path in the event of multiple matches.

By the way, you will notice some quirks in how **Set-Location** behaves in older versions of PowerShell. For example, the following will produce an error in PowerShell version 1.

```
cd..
```

In Cmd.exe, that would move up one directory level to C:\; in PowerShell version 1 it generates an error because PowerShell needs a space between the command and any parameters.

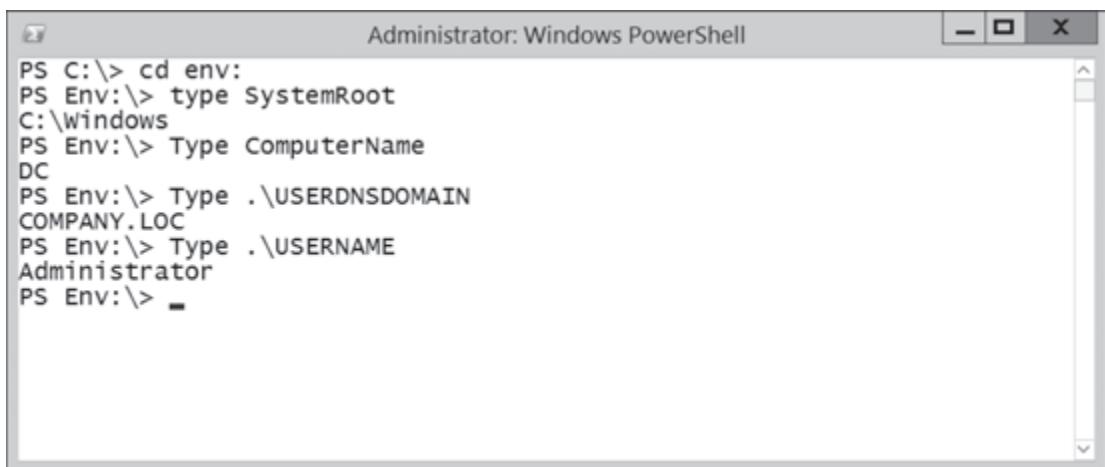
Do you know how to solve that? Start using PowerShell version 4 (also fixed in version 3)!

## Set-Location

Changing location is easy using the **Set-Location**, or its alias **CD**.

For example, try the following:

```
cd env:  
type SystemRoot
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history is as follows:

```
PS C:\> cd env:  
PS Env:\> type SystemRoot  
C:\Windows  
PS Env:\> Type ComputerName  
DC  
PS Env:\> Type .\USERDNSDOMAIN  
COMPANY.LOC  
PS Env:\> Type .\USERNAME  
Administrator  
PS Env:\> _
```

Figure 5-16

This uses **Set-Location** to change to the ENV: drive—the disk drive that contains all the environment variables on your computer. It then uses the **Type** alias—that's the **Get-Content** cmdlet, by the way—to retrieve the contents of the item named “systemroot.” In this case, that item is an environment variable and **Get-Content** displays its contents: C:\Windows. So, you've just learned a new cmdlet: **Get-Content**! That cmdlet has many parameters that customize its behavior, allowing you to filter the content as it is being displayed, read only a specified number of characters, and so forth. We won't be covering the cmdlet in any more depth right now, but feel free to look it up in PowerShell's Help, if you like.

## Cmdlets for working with items

PowerShell uses the word item to generically refer to the “stuff located in a PSDrive.” That means an item could be a file, a folder, a registry value, a registry key, a certificate, an environment variable, and so forth.

PowerShell has a variety of cmdlets for manipulating items:

- Copy-Item
- Clear-Item
- Get-Item
- Invoke-Item
- New-Item
- Move-Item
- Remove-Item
- Rename-Item
- Set-Item

Some of these will look familiar to you. For example, **Remove-Item** (whose alias is **Del**) is used to delete items, whether they are files, folders, registry keys, or whatever. The old **Move**, **Ren**, and **Copy** commands are now aliases to **Move-Item**, **Rename-Item**, and **Copy-Item**.

### Copy-Item

For example, here you can see a directory listing that includes a folder named Subfolder; you then use the **Copy-Item** cmdlet to create a copy of it named Newfolder.

```
dir -Directory
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "dir -Directory" is run, listing a single folder named "subfolder" at the path "C:\scripts". The "Mode" column shows "d----", "LastWriteTime" shows "6/30/2013 1:18 PM", and "Name" shows "subfolder". A new command "Copy-Item subfolder MyNewFolder" is then run, creating a new folder "MyNewFolder" in the same location.

```
Administrator: Windows PowerShell
PS C:\scripts> dir -Directory

 Directory: C:\scripts

 Mode                LastWriteTime      Length Name
 ----              - - - - - - - - - - - - - - - -
 d----       6/30/2013 1:18 PM           subfolder

PS C:\scripts> Copy-Item subfolder MyNewFolder
```

Figure 5-17

In this example, we will copy the folder named subfolder to MyNewFolder.

```
Copy-Item .\subfolder .\MyNewFolder
dir -Directory
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Copy-Item .\subfolder .\MyNewFolder" is run, followed by "dir -Directory". A table then displays the contents of the "C:\scripts" directory:

Mode	LastWriteTime	Length	Name
d----	6/30/2013 1:30 PM		MyNewFolder
d----	6/30/2013 1:18 PM		subfolder

Figure 5-18

The **Copy-Item** cmdlet is incredibly powerful. It supports a **-Recurse** parameter, which allows it to work with entire trees of objects and it supports the **-Include** and **-Exclude** filtering parameters, as well as **-Filter**. For example, the following will copy all files with a .PS1 filename extension to a folder named Newfolder. However, it will not copy files matching the wildcard pattern Demo?.ps1.

```
copy *.ps1 newfolder -Exclude demo?.ps1
```

This cmdlet also supports the **-WhatIf** and **-Confirm** parameters introduced earlier.

## More item cmdlets

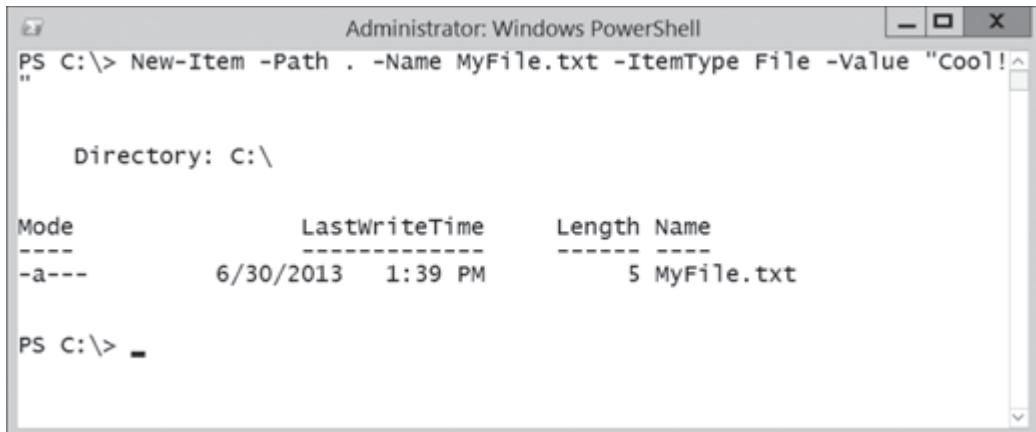
The **Move-Item** cmdlet supports a similar set of functionality. Even **Rename-Item** supports the **-WhatIf** and **-Confirm** parameters, so that you can test what it's doing before actually committing yourself.

**Clear-Item** works similarly to **Remove-Item**. However, it leaves the original item in place and clears out its contents, making it a zero-length file. That might not seem useful with files and folders, but it's definitely useful with other **PSDrive** providers, such as the registry, where **Clear-Item** can eliminate the value from a setting but leave the setting itself intact. Similarly, **Set-Item** might not seem to have any use in the file system, but it's useful for changing the value of registry settings.

## New-Item

Last up is **New-Item**, which, as you might guess, creates an all new item. Of course, you will need to tell PowerShell what kind of item you would like to create, and that type of item must match the drive where the item will be created. You can even create new registry entries! For example, the following command will create a new file and place some text in it.

```
New-Item -Path . -Name MyFile.txt -ItemType File -Value "Cool!"
```



```
Administrator: Windows PowerShell
PS C:\> New-Item -Path . -Name MyFile.txt -ItemType File -Value "Cool!"
"
Directory: C:\

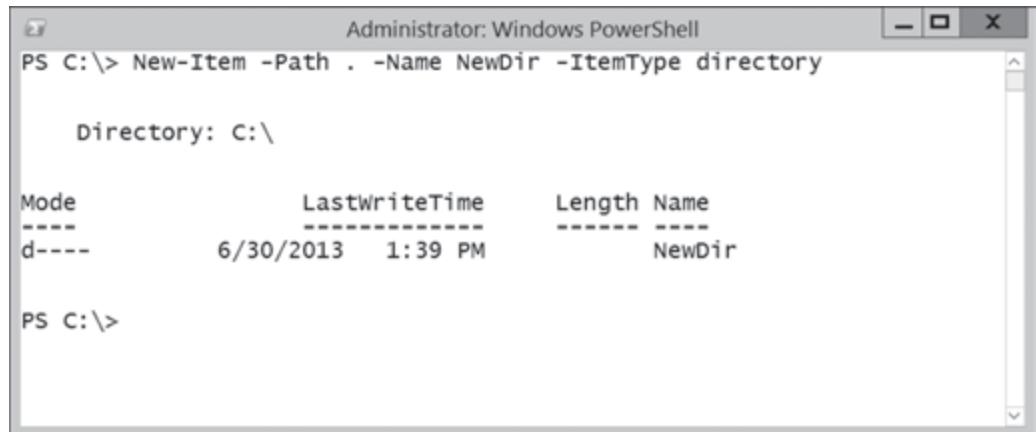
Mode          LastWriteTime    Length Name
----          -----          ---- 
-a---  6/30/2013 1:39 PM      5 MyFile.txt

PS C:\> _
```

Figure 5-19

The **-Path** parameter indicates where the system should create the item and the remaining parameters specify its name, type, and initial contents. You might also specify “directory” to create a new directory—and, by the way, you’ve just found the cmdlet that’s used instead of the old **MkDir** command! Go ahead, try the alias **Mkdir!** To create a new folder using **New-Item**, you must specify the **-ItemType** directory parameter.

```
New-Item -Path . -Name NewDir -ItemType directory
```



```
Administrator: Windows PowerShell
PS C:\> New-Item -Path . -Name NewDir -ItemType directory

Directory: C:\

Mode          LastWriteTime    Length Name
----          -----          ---- 
d---  6/30/2013 1:39 PM      NewDir

PS C:\>
```

Figure 5-20

You don’t have to include parameter values like “file” in quotation marks, unless the value contains a space. It doesn’t hurt to enclose them in quotation marks, though, and it’s not a bad habit to get into, because quotes will always work, even if the value contains spaces.

Use PowerShell’s built-in Help to explore some of the other options available to these cmdlets and you’ll soon be working with all types of items from the various PSDrives available to you.

## Exercise 5 – Working with providers

Time: 30 minutes

The PowerShell team wanted to make your transition to working with the file system as easy as possible. The inclusion of the aliases from both the \*nix and CMD.exe world helps get started immediately. The downside to using the aliases at first is that many admins don't realize that there are cmdlets behind the alias—and that the new cmdlets provide many more features than the old commands.

With this in mind, the goal of this exercise is to solve the tasks without ANY aliases. That's right, not even **CD** or **Dir**. Take time during this exercise to look at the Help for the cmdlets and discover additional features that may interest you.

Remember: no aliases this time.

### Task 1

Create the following folder structure of the root of your C: drive:

C:\Scripts  
C:\Scripts\TextFiles  
C:\Scripts\LabFiles

### Task 2

Create new text files in the following locations. You can name the files anything you like, but make sure to add some text to them when you create them.

C:\Scripts\MyText.txt  
C:\Scripts\Test.txt  
C:\Scripts\TextFiles\MoreText.txt  
C:\Scripts\TextFiles\MoreText2.txt

### Task 3

Copy the folder c:\Scripts\TextFiles to c:\scripts\labfiles.

### Task 4

If you have a remote computer available, can you copy the folder to the remote computer without mapping a drive?

### Task 5

If you have a remote computer, use **PSDrive**, **Net Use**, and **New-SMBMapping** to map drive letters F:, G:, and H: to the root of the remote computer. You can use the UNC \\Server\C\$ or any other share that might be on the remote computer.

## Task 6

Create a registry key of <yourName> in HKCU::

## Task 7

Move the folder c:\scripts\Labfiles to c:\LabFiles.

## Task 8

Find a list of all the drives that PowerShell recognizes on your system.

## Task 9

Can you find a list of the environment variables for your system?

## Chapter 6

# Pipelines for rapid management

### What does the pipeline do anyway?

Perhaps the most powerful concept in Windows PowerShell is its rich, object-oriented pipeline. You may already be familiar with pipelines from the Cmd.exe console or from MS-DOS. For example, one common use was to pipe a long text file to the **More** utility and create a paged display of text.

```
type myfile.txt | more
```

This technique evolved directly from \*nix shells, which have used pipelines for years to pass, or pipe, text from one command to another. In PowerShell, this concept takes on whole new meaning as cmdlets work with rich objects rather than text, and pipe those objects to one another for great effect. For example, consider the following cmdlet.

```
get-process
```

Administrator: Windows PowerShell							
Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	ID	ProcessName
107	11	3076	11540	104	0.83	1548	conhost
42	5	724	3284	48	0.02	2688	conhost
212	11	1620	3864	47	0.11	344	csrss
168	15	1756	11648	76	0.41	420	csrss
334	31	13976	18412	719	0.16	1324	dfsrs
102	8	1416	3900	22	0.02	1560	dfssvc
194	13	3172	10260	48	0.16	1624	dllhost
5251	3692	47460	47316	94	0.38	1356	dns
206	17	23636	36112	127	0.61	812	dwm
917	49	17992	47288	308	1.09	324	explorer
0	0	0	24	0		0	Idle
89	11	1556	4428	28	0.02	1384	ismserv

Figure 6-1

Run that and you will see a list of processes. It's easy to assume that what you're seeing is the actual output of the cmdlet; that is, it's easy to think that the cmdlet simply produces a text list. **But it doesn't.** The cmdlet produces a set of process objects. Since there's nothing else in the pipeline waiting for those objects, PowerShell sends them to the default output. That cmdlet calls one of PowerShell's formatting cmdlets, which examines various properties of the objects—in this case, the **Handles**, **Nonpaged Memory(NPM)**, **Paged Memory(PM)**, **Working Set(WS)**, **Virtual Memory(VM)**, **CPU**, **ID**, and **ProcessName**—and creates a table of information. So, the result that you see is certainly a text list, but you're not seeing the intermediate steps that created that text list from a collection of objects.

### By the way...

How does PowerShell decide what properties to use when it converts objects into a text-based listing? It's not arbitrary. For most object types, Microsoft has pre-defined the “interesting” properties that PowerShell uses. These pre-defined formats are in a file called DotNetTypes.format.ps1xml, located in the PowerShell installation folder. Other pre-defined types are defined in other files, such as FileSystem.format.ps1xml, Certificate.format.ps1xml, and so forth. These files are digitally signed by Microsoft, so you can't modify them unless you're prepared to re-sign them using your own code-signing certificate. However, you can build custom formats, which are discussed later.

So, what in fact does the pipeline do? It allows you to connect cmdlets together, passing objects from one to the next, to create powerful solutions. As you proceed through this chapter, and the rest of the book, you will learn more and more about using the pipeline—and that's really what most of this book is about. Let's start by examining what an object is and why we would even want to pass it.

## Discovering object types, methods, and properties

Many IT pros just starting out with PowerShell grapple with the concept of an object. If you have previously scripted or programmed in other languages, you may already be comfortable with this concept.

Objects are real things that have properties that describe them and methods that control the object's behavior. Look around you right now, so you see your computer? It's an object and it has properties, such as processor type, amount of memory, and disk space. What can you do with your laptop? You can turn it on, you can type on the keyboard, and you can launch and run applications. While this is a simple

example, objects are real things and Windows is full of them. When you run a PowerShell cmdlet, the cmdlet returns objects.

## Discovering objects

Using previous scripting languages, such as VBScript, were challenging due to the lack of information about an object, and the methods and properties that it contained. In fact, without a book or website such as MSDN explaining all the possible properties and methods, many scripters felt lost and had no idea how to use an object.

PowerShell is based on .NET, and it inherited an ability from the framework, called Reflection. Just like looking into a mirror, an object can describe itself and list its methods and properties.

To discover all the information about an object, including its methods and properties, instantiate the object and pass it to the cmdlet **Get-Member**.

```
Get-Service -name bits | Get-Member
```

Name	MemberType	Definition
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDependedOn
Disposed	Event	System.EventHandler Disposed(System....)
Close	Method	void Close()
Continue	Method	void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef Creat...
Dispose	Method	void Dispose(), void IDisposable.Dis...
Equals	Method	bool Equals(system.Object obj)
ExecuteCommand	Method	void ExecuteCommand(int command)
GetHashCode	Method	int GetHashCode()

Figure 6-2

**Get-Member** returns the object type (**TypeName**)—in this case **System.ServiceProcess**.

**ServiceController**—and the members of the object, both properties and methods. For our needs, any **MemberType** that has the word “property” in it is a property that can contain descriptive information about the object. There are several property types, such as **AliasProperty**, **ScriptProperty**, and just plain **Properties**, but for our purpose, they all work the same way.

You will also notice the different methods—things that the object can do. Right now, let’s focus on the properties and use the pipeline to help us select just the information that we want to see.

By the way, you won’t often see PowerShell admins use the full **Get-Member** cmdlet; instead you will usually see them use the alias **GM**. So, when you want to know more information about an object, “pipe to GM.” Here are a few examples that you can try.

```
Get-Process | GM
Get-NetIPAddress | GM
Get-History | GM
```

Sometimes, a cmdlet may produce several different object types. You will notice this by examining the **TypeNames** of the different objects. Here's an example for you to try that looks at the **System Eventlog** where the log contains several different types of objects. Try this and notice all the object types—along with their respective members (properties and methods).

```
Get-EventLog -LogName System | GM
```

Now, let's start making this useful.

## Selecting just the right information

**Select-Object** (or its alias, **Select**) selects properties that you specify. This cmdlet has very powerful parameters and takes a bit of work to understand. First, let's look at what we get when we pass a particular object to **Get-Member** (or its alias, **GM**), to see what properties and methods the object has.

```
Get-Process | GM
```

Name	MemberType	Definition
Handles	AliasProperty	Handles = HandleCount
Name	AliasProperty	Name = ProcessName
NPM	AliasProperty	NPM = NonpagedSystemMemorySize
PM	AliasProperty	PM = PagedMemorySize
VM	AliasProperty	VM = VirtualMemorySize
WS	AliasProperty	WS = WorkingSet
Disposed	Event	System.EventHandler Disposed(System...
ErrorDataReceived	Event	System.Diagnostics.DataReceivedEve...
Exited	Event	System.EventHandler Exited(System....

Figure 6-3

Examine all the properties that **Get-Process** provides about processes running on your computer. Perhaps you only want to see the properties for Name, VM, and CPU. You can use **Select-Object**, with the **-Property** parameter, to filter only those properties in your display.

```
Get-Process | Select-Object -Property Name, VM, CPU
```

Name	VM	CPU
conhost	107479040	0.140625
conhost	50180096	0
csrss	49254400	0.171875
csrss	83906560	1.140625
dfssrs	753561600	0.234375
dfssvc	23175168	0.015625
dllhost	49864704	0.078125
dns	98938880	0.3125
dwm	140271616	0.921875
explorer	326488064	1.921875
idle	65536	
ismserv	32333824	0.03125

Figure 6-4

See? The **Select-Object** cmdlet selected only those properties we wanted. This is very handy for generating reports that contain only the information you need.

Before we go further, we should point out that most of the time we prefer to use the full cmdlet name, the parameter, followed by its arguments. This creates a very descriptive looking command that can be easily read, and if needed, easily examined in the Help file. Because **Select-Object** has an alias and position parameters (remember when you examined reading syntax earlier?), you can shorten this command, which is often done. Here is an example of the full command, followed by a shorter version that uses aliases.

```
Get-Process | Select-Object -Property name, VM, CPU, PS | Select name, VM, CPU
```

You can see the shortened version (elastic notation) is faster to type, but less readable. You should get used to seeing and using both. We prefer to use shortened commands when using PowerShell interactively (real-time) when managing systems. However, this is a bad practice when you are scripting and you should always use the more descriptive version. When you are first learning PowerShell, it's best to always use the full version, the full cmdlet name, and the parameter with its arguments—this will allow you to learn the parameters and always know exactly where in the Help file to check if you need more information.

**Select-Object** has parameters to control the amount of information that is displayed. In this example, you might only want the first two or last two processes listed. This becomes truly powerful later when we examine sorting data.

```
Get-Process | Select-Object -Property Name, VM, CPU -First 2
Get-Process | Select-Object -Property Name, VM, CPU -Last 2
```

```

Administrator: Windows PowerShell
PS C:\> Get-Process | Select-Object -Property Name, VM, CPU -First 2
Name                               VM          CPU
----                               --          ---
conhost                           107479040   0.4375
conhost                           50180096    0

PS C:\> Get-Process | Select-Object -Property Name, VM, CPU -Last 2
Name                               VM          CPU
----                               --          ---
wininit                           41656320   0.0625
winlogon                          54116352   0.203125

```

Figure 6-5

## Objects after Select-Object

**Select-Object** is very useful in controlling the specific information, and amount of information, that you want to see. The objects are left intact as you continue to use the pipeline, so further cmdlets can be used to work with them.

```
Get-Service | Select-Object -Property Name, status | gm
```

```

Administrator: Windows PowerShell
PS C:\> Get-service | select-object -Property Name, status | gm

TypeName: Selected.System.ServiceProcess.ServiceController
Name      MemberType  Definition
----      --          --
Equals    Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType   Method     type GetType()
ToString  Method     string ToString()
Name      NoteProperty System.String Name=ADWS
Status    NoteProperty System.ServiceProcess.ServiceControllerStatus Status...

PS C:\>

```

Figure 6-6

Notice that the object type (**TypeName**) is still maintained. The results of this command (without the **| GM**) still returns **ServiceController** objects, but only displaying the properties you request.

## When you only want a value

As you will see later in this chapter, there are times when you don't want the original object type; instead, you just want the results from a property in its native data type, such as string or integer. **Select-Object** has the **-ExpandProperty** parameter, which will extract only the values inside the property.

```
Get-service | Select-Object -ExpandProperty name
```

```
Administrator: Windows PowerShell
PS C:\> Get-service | select-object -ExpandProperty name
ADWS
AeLookupsvc
ALG
AppIDSvc
Appinfo
AppMgmt
AppReadiness
AppXsvc
AudioEndpointBuilder
Audiosrv
BFE
BITS
BrokerInfrastructure
Browser
CertPropSvc
```

Figure 6-7

In Figure 6-7, if you pipe this to **Get-Member**, you will see that it is no longer a **ServiceController** object, but a string. The extracted values of the **Name** property in the original data type have been retrieved. This technique will be very useful later in this chapter.

## Executing methods on an object

Let's take a moment and look at executing methods on an object before returning back to properties. We still have a lot of work to do with properties and the power of the pipeline, but it will be worth the detour.

First, you should know that there are several ways to do this and you will learn them as you go through the rest of the book. One way to execute a method on an object is to use the **ForEach-Object** cmdlet. Let's work through a scenario and build up to using the **ForEach-Object** cmdlet along the way.

Let's start with a simple command that will list the status of the **bits** service.

```
Get-Service -Name bits
```

When you run this, the service status is either stopped or running, it really doesn't matter for this example. The reason we use the **bits** service for this demonstration is because it isn't harmful to play with on your computer. If the service is stopped, one way of starting the service is with the **Start-Service** cmdlet.

```
Start-Service -Name bits
```

You could also start the service by using the pipeline and piping the results of **Get-Service** to **Start-Service**, as shown in the following example.

```
Get-Service -Name bits | Start-Service
```

Pretend for a moment, that the **Start-Service** cmdlet didn't exist. If you wanted to start the service without having a cmdlet to do so, first examine the object for any methods that may produce the result you're looking for, by using **Get-Member**.

```
Get-Service -Name bits | gm
```

Name	MemberType	Definition
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDependedOn
Disposed	Event	System.EventHandler Disposed(System....)
Close	Method	void Close()
Continue	Method	void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef Creat...
Dispose	Method	void Dispose(), void IDisposable.Dis...
Equals	Method	bool Equals(System.Object obj)
ExecuteCommand	Method	void ExecuteCommand(int command)
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifetimeServ...
Pause	Method	void Pause()
Refresh	Method	void Refresh()
Start	Method	void Start(), void Start(string[] args)
Stop	Method	void Stop()
WaitForStatus	Method	void WaitForStatus(System.ServicePro...

Figure 6-8

If you examine the list of methods, you will notice that there is a **Stop** and **Start** method. If you don't know what these methods do, you could check on MSDN. In this case, they do exactly what you think—they start and stop a **ServiceController** object. If you want to start the service, use the **ForEach-Object** cmdlet.

The **ForEach-Object** cmdlet actually straddles a line between interactive use and scripting. Its alias, **ForEach**, can be seen as a scripting construct and we'll cover it as such later. However, when used as a cmdlet, its syntax is somewhat different, so we'll cover that part here.

In its simplest form, **ForEach-Object** accepts a collection of objects from the pipeline, and then executes a script block that you provide. It executes the script block once for each pipeline object and within the script block you can use the special `$_.variable` to refer to the current pipeline object. We haven't talked about variables yet, but this special two-character symbol is important now. It holds the current object passing the pipeline. In this example, we only will have one object, the **bits** service, but often you will have hundreds or thousands.

Here's how to start the **bits** service with **ForEach-Object**.

```
Get-Service -Name bits | ForEach-Object { $_.start() }
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `Get-Service -Name bits` is run, displaying a table with columns "Status", "Name", and "DisplayName". The service "bits" is listed as "Stopped". The command `Get-Service -Name bits | ForEach-Object {$_ .start()}` is then run, followed by another `Get-Service -Name bits` command. The second table shows the service "bits" is now "Running".

Status	Name	DisplayName
Stopped	bits	Background Intelligent Transfer Ser...

Status	Name	DisplayName
Running	bits	Background Intelligent Transfer Ser...

Figure 6-9

The command in Figure 6-9 pipes the **bits** service across the pipeline, where it is temporarily stored in the `$_.variable`. To access a method, or even a property, you follow the `$_.` with a period, and then select a method or property. If you want to execute a method, you must use the `()` at the end of the method. We will look at those `()` on methods more deeply later, but if you forget the `()`, the command will not execute.

If you want to check the status of the service using the same process, replace the method above with the property `status`.

```
Get-Service -Name bits | ForEach-Object { $_.status }
```

We will be diving more into methods later, but first, a few cautions about using **ForEach-Object**.

### Caution!

The **ForEach-Object** cmdlet is very powerful and it is very useful to execute methods when no other cmdlet exists to perform the task. Since **ForEach-Object** can cycle through all the objects passed to it, it can create great solutions and sometimes great problems. Examine the following command, but do not run it!

```
Get-Service |ForEach-Object { $_.Stop() }
```

Do you see what would happen if you accidentally ran this command? This would stop all services—well, technically not all of them, Windows will hang before all of them stop—and this is probably not the result you wanted. When using cmdlets that make changes to your system, or methods that affect the system, you need to fully understand the expected outcome before pressing **Enter**. We will examine how to determine this later in the chapter.

One other note of caution: the **ForEach-Object** cmdlet has two aliases that you will see from time-to-time. In fact, we use the first one often.

```
Get-Service -Name bits |ForEach { $_.status }
```

Notice that the alias **ForEach** can be used to shorten the command. This alias is still very descriptive of what the cmdlet will do.

There is another alias for **ForEach-Object** that we don't particularly like to use, but you will see it often. It's not as descriptive and can make your overall command harder to read. It's the **%** character.

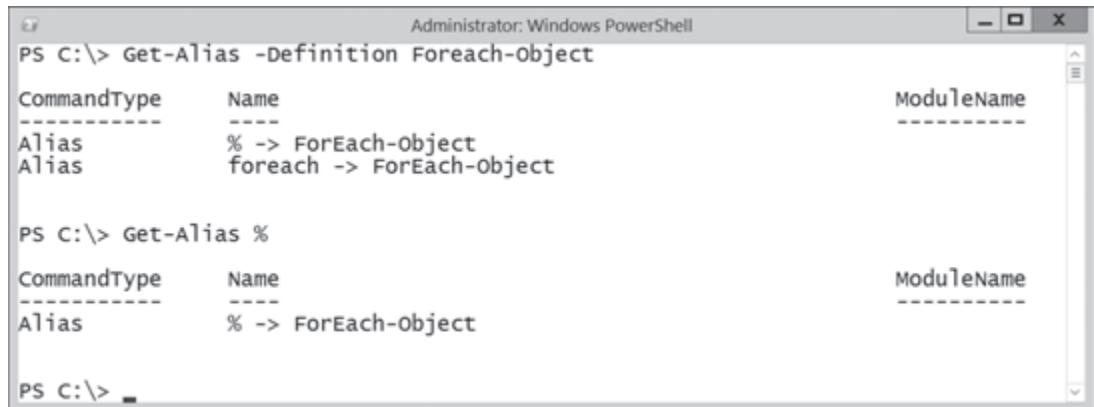
```
Get-Service -Name bits | % { $_.status }
```

This creates confusion for when you are just starting out with PowerShell and using it can lead to mistakes, so you should avoid it. We describe it here so you can recognize it when you see it on a blog or example. Remember, anytime you want to see a list of alias for a cmdlet, run the following:

```
Get-Alias -Definition Foreach-Object
```

or the reverse.

```
Get-Alias %
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-Alias -Definition Foreach-Object" is run, resulting in the following output:

CommandType	Name	ModuleName
Alias	% -> ForEach-Object	
Alias	foreach -> ForEach-Object	

Then, the command "Get-Alias %" is run, resulting in the following output:

CommandType	Name	ModuleName
Alias	% -> ForEach-Object	

Finally, the command "PS C:\> \_" is run.

Figure 6-10

Before we move on with more examples of using the pipeline and how to troubleshoot the pipeline, try a quick exercise to practice your new skills!

# Exercise 6A – Working with providers

Time: 20 minutes

In this exercise, you'll use **Get-Member** to learn more about the objects produced by various cmdlets. To help review previous concepts, you will need to use the **Get-Help** cmdlet to find cmdlets, and then examine the properties and methods of those objects. You will also have a chance to use the **Select-Object** cmdlet to grab specific properties.

## Task 1

Can you discover a cmdlet that will retrieve the current IP address information for your computer?

## Task 2

Using the cmdlet above, what type of object does it produce?

## Task 3

Using the cmdlet above, is there a property that will display only the IP address?

## Task 4

Using the **Select-Object** cmdlet and the cmdlet you discovered above, display the **IPAddress** and **Interface** aliases.

## Task 5

Identify a cmdlet that will display the current date and time.

## Task 6

What type of object does the cmdlet from the above task produce? Can you select only day of the week from the results?

## Task 7

Discover a cmdlet that will allow you to examine the event logs. Display the security event log.

## Task 8

Using the cmdlet discovered above, from the system log, display only ten of the newest events that are errors. Select only the properties for the time, source, and message.

## Task 9

What type of object does **Get-Service** produce? Is there a method that will pause a service?

## Task 10

Is there a method that will stop or kill a process?

## Task 11

Start Notepad. Using **Get-Process** and a method, stop the Notepad process. What other ways can you stop a process?

## Understanding how it all works

Before you dive into the rest of the book and start building amazing one-liners and using the pipeline, it's important to understand how the pipeline works. You need to be able to troubleshoot the pipeline when you don't receive the results you expect and you need to become more efficient at building one-liners that work the way you want.

You will only become highly effective with PowerShell if you take the time now to understand what is happening and how to read deeper into the syntax and Help files of cmdlets.

We have split the process of how the pipeline works into four parts, each with an example to explore what is happening. During this process, note that we never actually run the pipeline commands. This is kind of the point—you can figure out if something will work before you even run it. Also, you will understand why something doesn't work—and how to fix it when you do run a pipelined command.

The tools you need are **Get-Help** and **Get-Member**. By using the Help files and information about an object, you have everything you need.

## What you already know

You already know that objects are passed through the pipeline. These objects can be viewed in detail by using **Get-Member**. When objects are passed across the pipeline to other cmdlets, the cmdlet receiving the object has to be able to determine how to handle it.

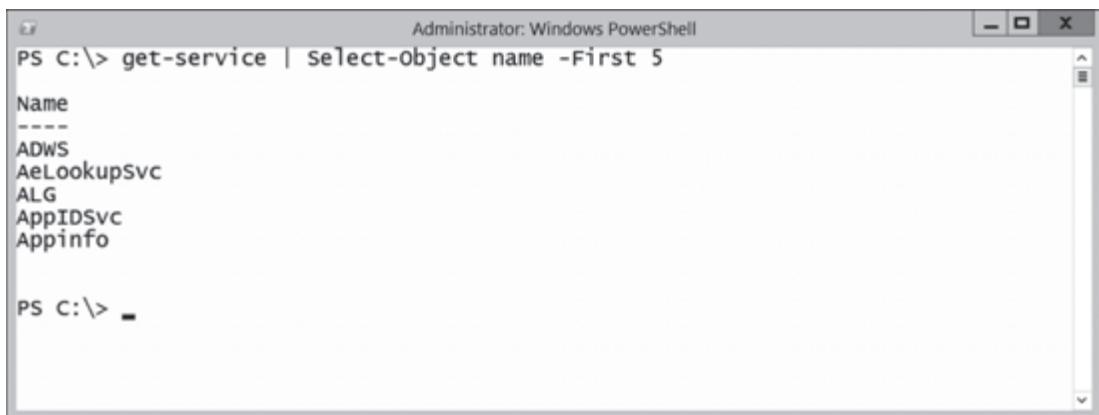
---

### Note:

Remember, not all cmdlets can actually receive objects from the pipeline. Don't worry, you can still solve your problem and we will demonstrate how to do that, last. Think of it as a last resort option.

When you use a cmdlet that gathers more than one object, known as a collection of objects, the pipeline will pass them one at a time to the next cmdlet in line.

```
get-service | Select-Object name -First 5
```



```
Administrator: Windows PowerShell
PS C:\> get-service | select-object name -First 5
Name
-----
ADWS
AeLookupSvc
ALG
AppIDSvc
Appinfo

PS C:\> -
```

Figure 6-11

**Get-Service** gathered all the service objects on our computer. One object at a time was passed to the **Select-Object** cmdlet, where we selected the **Name** property. In our example, we limited the results to the first five, but the result is the same—one object is passed at a time.

It just so happens that the **Select-Object** cmdlet knew how to accept an object from **Get-Service**. The question is: how did it know?

## How objects are passed

When an object is passed to a receiving cmdlet, the object must be bound to a parameter that can receive (or grab) the object. Think about that for a moment. The cmdlet receiving the object attaches the object to one of its parameters. Once attached, the receiving cmdlet can then perform its action on that object.

To see which parameters might possibly receive the object, use **Get-Help** with the **-Full** parameter.

```
Get-Help Stop-Service -full
```

```

Administrator: Windows PowerShell

-Include <String[]>
    Stops only the specified services. The value of this parameter
    qualifies the Name parameter. Enter a name element or pattern, such as
    "s*". Wildcards are permitted.

    Required?           false
    Position?          named
    Default value
    Accept pipeline input?   false
    Accept wildcard characters? true

-InputObject <ServiceController[]>
    Specifies ServiceController objects representing the services to be
    stopped. Enter a variable that contains the objects, or type a command
    or expression that gets the objects.

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

```

Figure 6-12

Notice in the **Parameter** section that the **-Full** switch has exposed some additional information. Look specifically at the two parameters **-Include** and **-InputObject** and notice the question “Accept pipeline input?” **-Include** does not accept pipeline input—it will not accept an object across the pipeline. However, **-InputObject** does accept pipeline input. Also you will notice **how** the parameter accepts pipeline input—in this case, **ByValue**.

There are two methods of matching objects from the sending cmdlet across the pipeline to the receiving cmdlet’s parameters, **ByValue** and **ByPropertyName**. **ByValue** is the first way the cmdlet will try and it’s the preferred method. If that fails, the cmdlet will then try **ByPropertyName**. If that fails, well...we will get to that. Over the next few pages, let’s see how each of these works.

## Parameter binding ByValue

It is easier to learn about the pipeline through a series of examples. Now keep in mind, some of these examples are not safe to run on your computer and that's not the intent. The goal is to figure out what will happen without running the one-liner.

The first way a cmdlet will try to accept an object is by using **ByValue**. If this works, then the cmdlet will act on the object. Consider the following example (remember—*do not* attempt to run this):

```
Get-Service | Stop-Service
```

**Get-Service** gathers all the service objects and sends them across the pipeline. Let's refer to this as the sending cmdlet. **Stop-Service** receives the objects and determines if it can use them—this is the receiving cmdlet. We will note this as:

```
Get-Service | Stop-Service
  Sending    | Receiving
```

So, the first task is to determine the type of object that the sending cmdlet is trying to send across the pipeline. You did this earlier in the chapter, by using **Get-Member**. Here's how to determine the object type.

```
Get-Service | gm
```

Name	MemberType	Definition
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDependedOn
Disposed	Event	System.EventHandler Disposed(System....)
Close	Method	void Close()
Continue	Method	void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef Creat...
Dispose	Method	void Dispose(), void IDisposable.Dis...
Equals	Method	bool Equals(system.Object obj)
ExecuteCommand	Method	void ExecuteCommand(int command)

Figure 6-13

The **TypeName** (object type) is **System.ServiceProcess.ServiceController**. For our purposes, we only really need to know the last part after the last period—**ServiceController**. This is the type of object being sent, the next task is to determine if the receiving cmdlet (**Stop-Service**) will accept this object type by using **ByValue**.

This information is stored in the Help file for the receiving cmdlet. Remember, to see this information you need to use the **-Full** switch.

```
Get-Help Stop-Service -Full
```

Look for any parameters that accept pipeline input by using **ByValue**. In this case, there are two parameters that accept pipeline input by using **ByValue**: **-InputObject** and **-Name**.

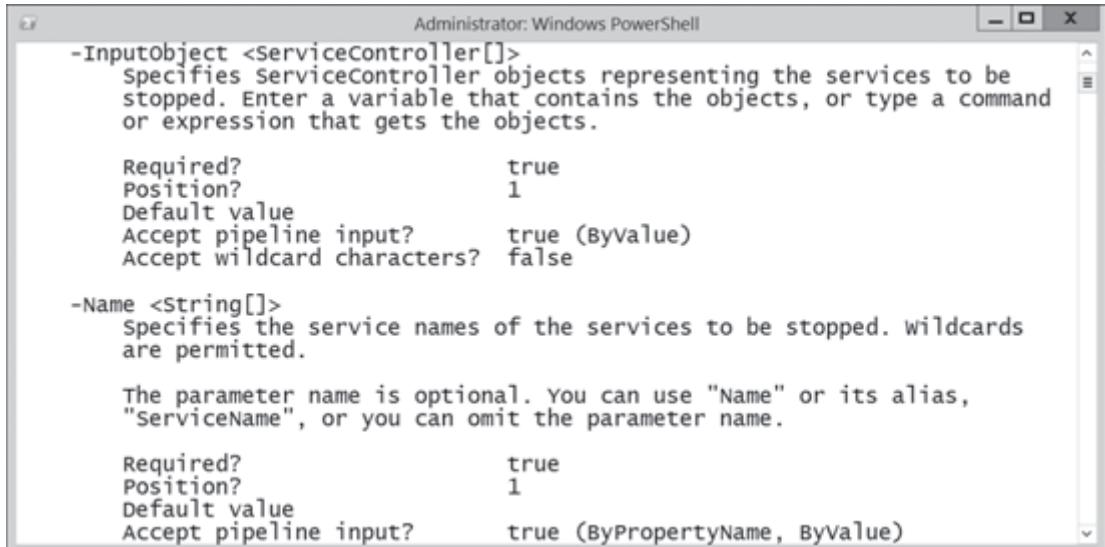


Figure 6-14

Notice the **Value** type of the argument for the two parameters. **-Name** has a value type of **String**—this is the type of object it accepts. We don't have a string object, we have a **ServiceController** object.

Ah! Now look at **-InputObject**. It has an argument that accepts objects with a value type of **ServiceController**. This is the type of object we are sending to it—so **-InputObject** is the parameter that will accept our objects from **Get-Service**.

So, here is what we learned. **Get-Service** sends **ServiceController** objects across the pipeline. It just so happens that **Stop-Service** has a few parameters that accept pipeline input **ByValue** and one of the parameters (**-InputObject**) has a value match to **ServiceController**. What is the result of all of this? If you were to run this one-liner—it would work.

By the way, most of the time, cmdlets that have the same noun will generally pass objects by using **ByValue**.

## Parameter binding ByPropertyName

Passing objects by using **ByValue** is the first method a receiving cmdlet will try—but what if this fails? We have a backup plan, called **ByPropertyName**. This is a little more complex, but it's always good to have a backup plan. Take a look at the following:

```
Get-Service | Stop-Process
  Sending    | Receiving
```

Notice that the nouns don't match. This doesn't mean that we can't pass **ByValue**, it just means that the cmdlets probably won't work this way. To make sure, you should always check to see if it would work, by using **ByValue**. The first task is to determine the type of object being sent across the pipeline.

```
Get-Service | GM
```

Just as the last example, it's still a **ServiceController** object, but that's only because we used the same cmdlet as in the last example. So, the next task is to determine if **Stop-Process** will accept the **ServiceController** object **ByValue**.

```
Get-Help Stop-Process -Full
```

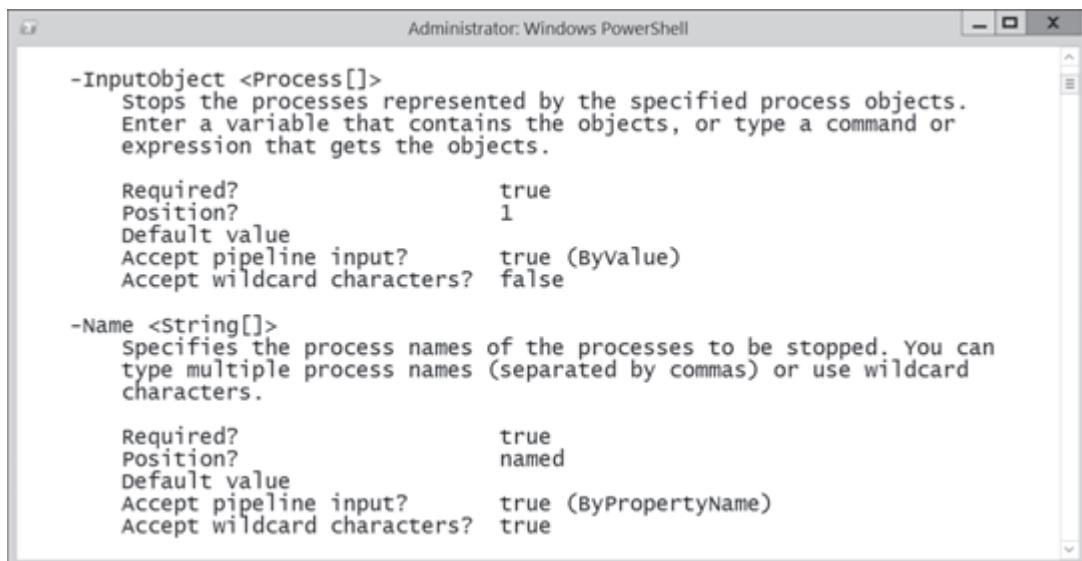


Figure 6-15

It turns out that there is a parameter that accepts pipeline input **ByValue**. It's the **-InputObject** parameter, but notice that it supports objects of the type **Process**. We are sending an object type of **ServiceController** when we used **Get-Service**. Since the receiving parameter only understands **Process** objects and not **ServiceController** objects, this will not work. So, **ByValue** fails. Time to move on to the backup plan.

For the backup plan, we check for parameters that accept pipeline input—but this time we use **ByPropertyName**. Once again, this information is in the Help file for the receiving cmdlet.

```
Get-Help Stop-Process -Full
```

When you examine the parameters, you will find two that accept pipeline input by using **ByPropertyName**: **-Name** and **-ID**. **-Name** accepts arguments of value type **String** and **-ID** accepts object that are integer.

Here is where the complex and confusing part comes into play in our backup plan. What you need to determine at this point is if there are any properties on the object being sent across the pipeline that match those two parameters. In other words, does the **ServiceController** object have a property called **Name** or **ID**? Remember, to determine what properties an object has: pipe to GM.

```
PS C:\> get-service | GM
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "PS C:\> get-service | GM" is entered. The output displays the members of the System.ServiceProcess.ServiceController type. The table has three columns: Name, MemberType, and Definition. The Name column lists methods and properties like Name, RequiredServices, Disposed, Close, Continue, CreateObjRef, Dispose, Equals, and ExecuteCommand. The MemberType column shows types like AliasProperty, Event, Method, and System.Runtime.Remoting.ObjRef. The Definition column provides the full definition for each member, such as "Name = ServiceName" for Name.

Name	MemberType	Definition
---	-----	-----
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDependedOn
Disposed	Event	System.EventHandler Disposed(System....)
Close	Method	void Close()
Continue	Method	void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef Creat...
Dispose	Method	void Dispose(), void IDisposable.Dis...
Equals	Method	bool Equals(system.object obj)
ExecuteCommand	Method	void ExecuteCommand(int command)

Figure 6-16

It turns out that there is a property called **Name**, but there is no property called **ID**. This is all the information we need. Now, for the confusing part.

Believe it or not, **Get-Service | Stop-Process** will work. See, the property of the object matches the name of the parameter that accepts **ByPropertyName**. If the parameter name and property names match, then the receiving cmdlet knows what to do with the object.

Also, the property **Name** on the object has a value type of **String**—which also matches the value type that the **-Name** parameter on **Stop-Process** accepts.

Now, you may be wondering, “OK, the **Name** property on the **ServiceController** object is giving its string values to the **-Name** parameter on **Stop-Process**. Doesn’t the **-Name** parameter expect to receive process names, not service names?” (You might not actually be asking that question—but it’s more fun this way.) The answer is Yes! We are sending the service name to **Stop-Process**, which will try to stop processes that have the names we sent. The one-liner works; it just doesn’t produce the results we would

expect. In fact, you might notice something interesting on your computer, if you want to try the following example.

```
Get-Service | Stop-Process -WhatIf
```

Notice that we are using the **-WhatIf** switch parameter, so that the cmdlet will tell us what it would do, but not actually perform the operation.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered was "Get-Service | Stop-Process -WhatIf". The output displays two error messages from the Stop-Process cmdlet:

- The first error message is: "Stop-Process : Cannot find a process with the name "iphlpsvc". Verify the process name and call the cmdlet again." It points to the line "+ Get-Service | Stop-Process -whatIf" and the word "Stop-Process".
- The second error message is: "Stop-Process : Cannot find a process with the name "Kdc". Verify the process name and call the cmdlet again." It also points to the same line and word.

Below the errors, the command "+ Get-Service | Stop-Process -WhatIf" is shown again, indicating the command was run without actually stopping the processes.

Figure 6-17

You will receive a lot of read-error messages telling you what is happening. **Stop-Process** can't find any processes that have service names, but if you look closely at Figure 6-17, on our computer it did! Sometimes a service starts a process with the same name, so in this case it would have killed the process!

This isn't a practical example, but it does show you the process of how an object can still cross the pipeline even if the receiving cmdlet doesn't support it **ByValue**. If you use cmdlets with nouns that don't match, this is usually how the process works.

What happens if the property name on the object doesn't match the parameter we want to send it to? Just change it! That's next.

## When **ByPropertyName** and Properties don't match

Often when an object is passed, there are parameters on the receiving cmdlet that could take our object and function as we would like, but the property name on the sending object doesn't match. Examine the following example.

```
Get-ADComputer -Filter * | Get-Process
  Sending      | Receiving
```

In this example, our goal is to receive a process list from every computer in Active Directory. **Get-AdComputer** is a cmdlet that is available on Domain Controllers or from the Remote Server Administration Tools. It has a mandatory parameter, **-Filter**, which in this case we are specifying with a wildcard character. Figure 6-18 shows what the result looks like when you run the command.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "PS C:\> Get-ADComputer -Filter \*" is entered at the prompt. The output displays properties for a single computer object:

```
DistinguishedName : CN=SAPIENDC,OU=Domain Controllers,DC=Company,DC=PRI
DNSHostName     : SAPIENDC.Company.PRI
Enabled          : True
Name             : SAPIENDC
ObjectClass      : computer
ObjectGUID       : 85e5a2e5-3bcc-4111-b3da-75911d3ae814
SamAccountName   : SAPIENDC$
SID              : S-1-5-21-3388612752-3846717619-3839454070-1001
UserPrincipalName :
```

Figure 6-18

Once again, we run through the process. The first task is to determine if the object can be passed by using **ByValue**, so you need to know the type of object you're passing.

```
Get-AdComputer -Filter * | GM
```

The type of object is an **ADCComputer** object. Well, check to see if the receiving cmdlet (**Get-Process**) accepts the **ADCComputer** object **ByValue**.

```
Get-Help Get-Process -Full
```

It turns out that while **Stop-Process** does have a parameter (**-InputObject**) that accepts pipeline input **ByValue**, but the object value type it accepts is called **Process**, not **ADCComputer**—so this doesn't work. Next, you should check for the backup plan—**ByPropertyName**.

```
Get-Help Get-Process -Full
```

Well, while you're checking the parameters for **ByPropertyName**, think about our goal again. We want to pass the names of the computers from **Get-AdComputer** to the parameter that accepts computer names on the **Get-Process** cmdlet. There is a parameter called **-ComputerName** that accepts the pipeline input **ByPropertyName** and the parameter **-Name** also accepts pipeline input.

This is where the problem occurs. Check the property on the object that contains the name of the computer.

```
Get-ADComputer -Filter * | GM
```

The name of the property is called **Name**. So if this object is passed to **Get-Process**, it will match up with the parameter called **-Name**—which is not what you want. What you really want is that the property on the **Get-ADComputer** object was called **ComputerName** rather than **Name**. Then it would match the **Get-Process** parameter **-ComputerName**!

Well, we can fix this by creating our own custom property on the **AdComputer** object. How? By using the **Select-Object** cmdlet.

```
Get-ADComputer -Filter * | Select @{ n = 'ComputerName'; e = { $_.Name } }
```

```
Administrator: Windows PowerShell
PS C:\> Get-ADComputer -Filter * | select @{n='ComputerName';e={$_.'Name'}}
ComputerName
-----
SAPIENDC

PS C:\> _
```

Figure 6-19

The unusual syntax you see is challenging at first, but you will have plenty of chances to use it and get more comfortable. It can be found in the Help file for **Select-Object** (see example 4 in Help for details), if you need help with the syntax.

```
@{ name = 'ComputerName'; expression = { $_.'Name' } }
```

In this code, you are creating a new property name called **ComputerName**. Then you are filling the property with values in the expression. Notice the **\$\_.**, which we talked about earlier. It contains the current object in the pipeline. The original object stored computer names in a property called **Name**. In the expression, we are asking for that property by using **\$\_.'name'**. The result is that we have created a new property called **ComputerName** with the values from the original name property. You can pipe this to **Get-Member** to see the new property, as well.

Did this help us? Yes! Now we are sending an object down the pipeline with a property name of **ComputerName**. The **Get-Process** cmdlet has a parameter called **-ComputerName** that accepts pipeline input (**ByPropertyName**). Perfect match!

```
Get-ADComputer -Filter * | Select @{ n = 'ComputerName'; e = { $_.name } } | Get-Process
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-ADComputer -Filter \* | select @{n='ComputerName';e={\$\_ . name}} | Get-Process". The output is a table with the following data:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	ID	ProcessName
54	7	1704	7264	57	0.02	2388	conhost
42	5	736	3244	48	0.00	2704	conhost
215	11	1468	3868	47	0.19	348	csrss
192	16	2136	13944	80	2.28	428	csrss
332	31	14496	19820	719	0.56	1324	dfsrs
101	8	1400	3892	22	0.02	1592	dfssvc
189	13	3124	10196	48	0.09	452	dllhost
5258	3695	48204	47644	97	0.31	1364	dns

Figure 6-20

This is a very important technique when passing objects down the pipeline and you will see us use it often. You will need it often as well, so we will make sure to have you practice it along with the syntax. There is only one more problem to solve: what if the cmdlet that you want to pass objects to doesn't accept *any* pipeline input? There is a last resort.

## The last resort: parentheticals

In this example, we want to retrieve the BIOS information from every computer on the network. In this case, we are going to use a cmdlet that we will discuss much more about later in the section on WMI—however it makes for a good example now. Without even knowing much about the cmdlet, you can still see how this is going to work.

```
Get-ADcomputer -filter * | Get-WmiObject -Class win32_bios
                           Sending      | Receiving
```

Once again, we start at the beginning of the process and see if the **Get-WMIObject** cmdlet will accept objects by using **ByValue**. From the last example, we know that the type of object we are sending already has an **ADComputer** object, so let's check the **Get-WMIObject** cmdlet for parameters that accept pipeline input.

```
Get-Help Get-WMIObject -Full
```

Whoa! There isn't a single parameter that accepts pipeline input! Many admins would just give up and believe they can't solve this problem—but you should know better. PowerShell was very well architected

and it can easily handle this problem. One thing you need to know is that you will run into this situation often—where the cmdlet doesn’t accept pipeline input or the parameter you want does not. Here’s how to fix it.

## It's all about parentheses

Just as in mathematics, PowerShell uses the order of operands, which means that whatever is inside of parentheses will be done first. You can try this at a PowerShell console.

```
(2+4)*(3-2)
```

So, let’s take this concept a little bit further. The **Get-WMIObject** has a parameter called **-ComputerName**. That’s the parameter that we want to fill with computer names from Active Directory. Here’s the concept, but this code doesn’t work yet.

```
Get-WMIObject win32_bios -ComputerName (Get-ADcomputer -Filter *)
```

See how we can take the results of a cmdlet and put them in as arguments to a parameter? This is a very useful technique, but the arguments must match the value type of the parameter. Let’s take a quick look at the parameter **-ComputerName** in the Help file.

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

Notice that the **-ComputerName** parameter needs arguments with a value type of **String**. That means that whatever we put into the parentheses must produce a string. In the following example, that is not the case.

```
Get-ADComputer -filter * | GM
```

Note the object type is **ADCComputer**—not a string. Also, if you run this command without the **| GM**, it will give us much more than just the computer name. First, let’s narrow down the data to only have the computer name.

```
Get-ADComputer -Filter * | Select -Property name
```

```

Administrator: Windows PowerShell
PS C:\> Get-ADComputer -Filter * | Select -Property name
name
-----
SAPIENDC

PS C:\> Get-ADComputer -Filter * | Select -Property name | gm

TypeName: Selected.Microsoft.ActiveDirectory.Management.ADComputer
Name      MemberType      Definition
----      -----          -----
Equals    Method          bool Equals(System.Object obj)
GetHashCode Method         int GetHashCode()

```

Figure 6-21

Using the **Select-Object** cmdlet narrowed down our results to just the computer name, but notice after piping it to **Get-Member** that it is still an **AdComputer** object, not the string that is expected by the **-ComputerName** parameter. You might remember at the beginning of this chapter we showed you a way to extract values from an object and put them in their original value types. This is the solution we need.

```
Get-ADComputer -Filter * | Select -ExpandProperty name
```

```

Administrator: Windows PowerShell
PS C:\> Get-ADComputer -Filter * | Select -ExpandProperty name
SAPIENDC
PS C:\> Get-ADComputer -Filter * | select -ExpandProperty name | gm

TypeName: System.String
Name      MemberType      Definition
----      -----          -----
Clone    Method          System.Object Clone(), System.Object ...
CompareTo Method         int CompareTo(System.Object value), i...
Contains Method         bool Contains(string value)
CopyTo   Method          void CopyTo(int sourceIndex, char[] d...
EndsWith Method         bool EndsWith(string value), bool End...
Equals   Method         bool Equals(System.Object obj), bool ...
GetEnumerator Method      System.CharEnumerator GetEnumerator()...

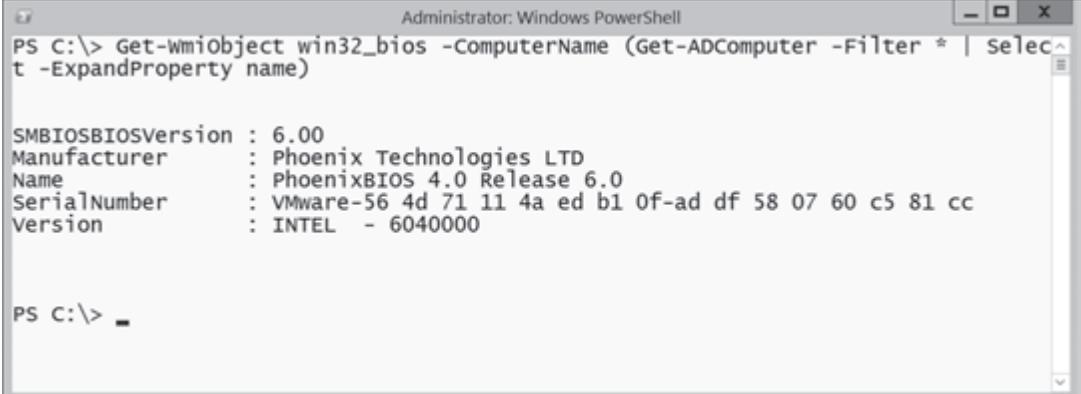
```

Figure 6-22

By using **Select-Object's** parameter **-ExpandProperty**, we can extract the computer names as their original **String** objects. Now, we just copy this into the parentheses of our **Get-WMIOBJECT** cmdlet.

```
Get-WmiObject win32_bios -ComputerName (Get-ADComputer -Filter * | Select -ExpandProperty name)
```

Windows PowerShell: TFM



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-WmiObject win32\_bios -ComputerName (Get-ADComputer -Filter \* | select -ExpandProperty name)". The output displays BIOS details for a specific computer:

```
SMBIOSBIOSVersion : 6.00
Manufacturer      : Phoenix Technologies LTD
Name              : PhoenixBIOS 4.0 Release 6.0
SerialNumber      : VMware-56 4d 71 11 4a ed b1 0f-ad df 58 07 60 c5 81 cc
Version           : INTEL - 6040000
```

PS C:\> -

Figure 6-23

We know that at first this type of solution is complex, but with practice it will become second nature. It's important because you will need it often. Remember, you can always find the solution you're looking for if you understand how the pipeline works and you know how to manipulate it.

Now, let's get some practice with this in the exercise.

# Exercise 6B – Working with providers

**Time:** 20 minutes

In this exercise, the goal is to determine how the pipeline works (or doesn't in some cases). The goal is not to run the full commands, but to use **Get-Help** and **Get-Member** to determine *if* they would work.

## Task 1

Determine exactly how the object will (or won't) be passed to the next command, either by using **ByValue** or **ByPropertyName**, and what parameter is making it possible.

```
Get-Service | Select-object -property name
```

## Task 2

Determine exactly how the object will (or won't) be passed to the next command, either by using **ByValue** or **ByPropertyName**, and what parameter is making it possible.

```
Get-Service | Format-Table -Property Name
```

## Task 3

Determine exactly how the object will (or won't) be passed to the next command, either by using **ByValue** or **ByPropertyName**, and what parameter is making it possible.

The following CSV has a column name **Names** and a list of computer names.

```
Import-Csv c:\Servers.csv | Get-Service -Name bits
```

## Task 4

Determine exactly how the object will (or won't) be passed to the next command, either by using **ByValue** or **ByPropertyName**, and what parameter is making it possible.

The following CSV has a column name **Names** and a list of computer names.

```
Import-CSV c:\servers.csv | Select -ExpandProperty Name | Get-Service -Name bits
```

## Task 5

Determine exactly how the object will (or won't) be passed to the next command, either by using **ByValue** or **ByPropertyName**, and what parameter is making it possible.

```
Get-Service -Name Bits -ComputerName (Import-Csv c:\servers.csv | Select -property name)
```

## Task 6

The following CSV has a column name **Names** and a list of computer names.

```
Get-Service -Name Bits -ComputerName (Import-Csv c:\servers.csv | Select -expandproperty name)
```

## Task 7

Create a CSV file named Servers.csv. Make two columns titled **Name** and **IPAddress**. Add one or two computer names and IP addresses. (Use the local host if you are working on your laptop.)

Create a pipeline solution that will get the **bits** service from those computers.

## Task 8

Using the same CSV file and a pipeline, can you retrieve the BIOS information from **Get-WmiObject -Class Win32\_bios?**

## Chapter 7

# Sorting and measuring data for output

### Sorting your results

Now that you've had a chance to dive deeply into the pipeline, it's time to start doing more with it. One of the benefits to the object-based nature of the pipeline is the ability to select and sort the specific information you want. You selected specific properties from objects in the last chapter by using **Select-Object**, and in similar fashion, you can sort those objects by using **Sort-Object**.

```
get-Process | Sort-Object -Property Handles
```

Administrator: Windows PowerShell							
Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	ID	ProcessName
0	0	0	24	0	0.02	0	Idle
32	5	580	2616	38	0.02	3052	VMToolsHookProc
43	5	736	3288	48	0.02	2844	conhost
53	2	276	1092	4	0.08	216	smss
56	7	1700	7152	57	0.17	1216	conhost
79	8	1304	5300	57	0.16	1504	VRService
81	8	760	3700	40	0.09	404	wininit
90	11	1552	4408	28	0.00	1376	ismserv
104	8	1432	3916	22	0.02	1584	dfssvc
109	11	1056	4404	21	0.03	1896	svchost
141	12	2044	8248	97	0.52	2836	TPAutoConnect
146	8	1668	7004	52	0.17	456	winlogon

Figure 7-1

In Figure 7-1, we've taken the output of **Get-Process** and sorted it by the **Handles** property. The default sort is ascending, but if you prefer, the cmdlet includes a **-Descending** parameter.

```
get-Process | Sort-Object -Property Handles -Descending
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
5253	3692	48072	47404	95	0.34	1344	dns
1276	151	57988	50780	1272	2.16	504	lsass
1135	48	13452	26824	131	1.23	836	svchost
911	49	17224	47068	299	0.81	2768	explorer
720	25	5864	11364	83	0.28	900	svchost
698	0	104	324	3	4.70	4	System
540	36	7124	15752	1180	0.22	980	svchost
535	37	31676	38324	577	0.81	1272	Microsoft.ActiveDi...
525	25	73060	77636	617	1.14	2432	powershell
438	19	11404	14284	62	0.17	800	svchost
401	25	4384	12108	81	0.25	1240	spoolsv
356	32	8464	10712	52	0.19	308	svchost

Figure 7-2

Remember that you are sorting on the properties of an object. This is an instance when **Get-Member** comes into play—to help you discover all the properties that an object contains. Anytime that you forget the name of a property or want to know the available properties that you can sort on, pipe the results to **Get-Member**.

If you take a look at the Help file for **Sort-Object**, you will notice that the **-Property** parameter accepts multiple values. In other words, you can have more than one sort. Notice in Figure 7-3 that we have sorted the output by status, but the names are not in alphabetical order.

```
Get-Service | Sort-Object -Property Status
```

Status	Name	DisplayName
Stopped	NtFrs	File Replication
Stopped	PerfHost	Performance Counter DLL Host
Stopped	pla	Performance Logs & Alerts
Stopped	NetTcpPortSharing	Net.Tcp Port Sharing Service
Stopped	vaultSvc	Credential Manager
Stopped	upnphost	UPnP Device Host
Stopped	TrustedInstaller	Windows Modules Installer
Stopped	RasAuto	Remote Access Auto Connection Manager
Stopped	RasMan	Remote Access Connection Manager
Stopped	UmRdpService	Remote Desktop Services UserMode Po...
Stopped	UIODetect	Interactive Services Detection
Stopped	PrintNotify	Printer Extensions and Notifications

Figure 7-3

Easy to fix! Notice that we added a second property to the sort.

```
Get-Service | Sort-Object -Property Status, Name
```

Status	Name	DisplayName
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Stopped	AppIDsvc	Application Identity
Stopped	Appinfo	Application Information
Stopped	AppMgmt	Application Management
Stopped	AppReadiness	App Readiness
Stopped	AppXsvc	AppX Deployment Service (Appxsvc)
Stopped	AudioEndpointBu...	Windows Audio Endpoint Builder
Stopped	Audiosrv	Windows Audio
Stopped	Browser	Computer Browser
Stopped	CertPropsvc	Certificate Propagation
Stopped	DefragSVC	Optimize drives

Figure 7-4

The ability to sort on multiple properties is very useful. When teaching PowerShell to admins, we often see a mistake that we don't want you to make. In Figure 7-4, we sorted on the status and the name of the service. The following does not produce the same results.

```
Get-Service | Sort-Object -Property Status | Sort-Object -Property Name
```

Status	Name	DisplayName
Running	ADWS	Active Directory Web Services
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Stopped	AppIDsvc	Application Identity
Stopped	Appinfo	Application Information
Stopped	AppMgmt	Application Management
Stopped	AppReadiness	App Readiness
Stopped	AppXsvc	AppX Deployment Service (Appxsvc)
Stopped	AudioEndpointBu...	Windows Audio Endpoint Builder
Stopped	Audiosrv	Windows Audio
Running	BFE	Base Filtering Engine
Running	BITS	Background Intelligent Transfer Ser...

Figure 7-5

PowerShell will do exactly what you tell it to do, even if it is wrong. In this case, we sorted by status, then re-sorted by name. This completely defeated the purpose. So remember to use **Sort-Object** once in your pipeline, and then use the multi-value parameter **-Property** if you want to sort on multiple properties.

## Displaying only unique information

Often, you will retrieve repetitive data but you will only want to see the unique entries. The **Sort-Object** cmdlet has a **-Unique** parameter that filters out repeating information. Here's a simple example.

```
$var = 1, 2, 2, 2, 3, 4, 4, 4, 5, 6, 7
$var | Sort-Object -Unique
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command \$var = 1, 2, 2, 2, 3, 4, 4, 4, 5, 6, 7 is run, followed by \$var | Sort-Object -Unique. The output shows the numbers 1 through 7 listed once each, indicating they are unique. The window has standard OS X-style scroll bars on the right side.

```
Administrator: Windows PowerShell
PS C:\> $var=1,2,2,2,3,4,4,4,5,6,7
PS C:\> $var | Sort-Object -Unique
1
2
3
4
5
6
7
PS C:\> $var=1,2,3,2,3,2,3,2
PS C:\> $var | Sort-Object -Unique
1
2
3
PS C:\> -
```

Figure 7-6

To demonstrate the **-Unique** parameter, we defined an array called **\$Var** with repeating numbers. We will discuss variables later, but notice how the results were sorted and how only one of each is displayed.

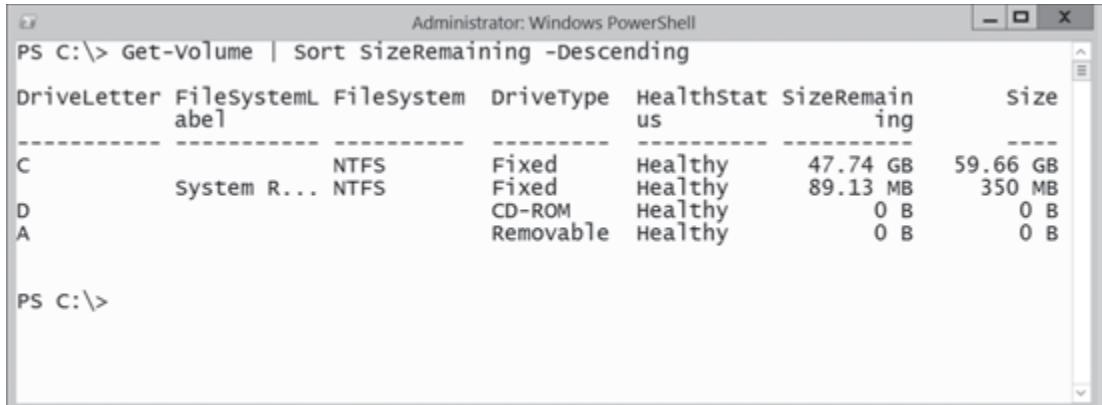
---

**Note:**

In our examples, we have been typing **Sort-Object**. However, like many cmdlets, it has an alias (**Sort**) and you will see it used often. In fact, the **-Property** parameter is positional and the parameter name is optional.

```
Get-Volume | Sort SizeRemaining -Descending
```

### Sorting and measuring data for output



Administrator: Windows PowerShell

```
PS C:\> Get-Volume | sort SizeRemaining -Descending
```

DriveLetter	FileSystemLabel	Filesystem	DriveType	HealthStatus	SizeRemaining	Size
C	System R...	NTFS	Fixed	Healthy	47.74 GB	59.66 GB
D		NTFS	Fixed	Healthy	89.13 MB	350 MB
A			CD-ROM	Healthy	0 B	0 B
			Removable	Healthy	0 B	0 B

```
PS C:\>
```

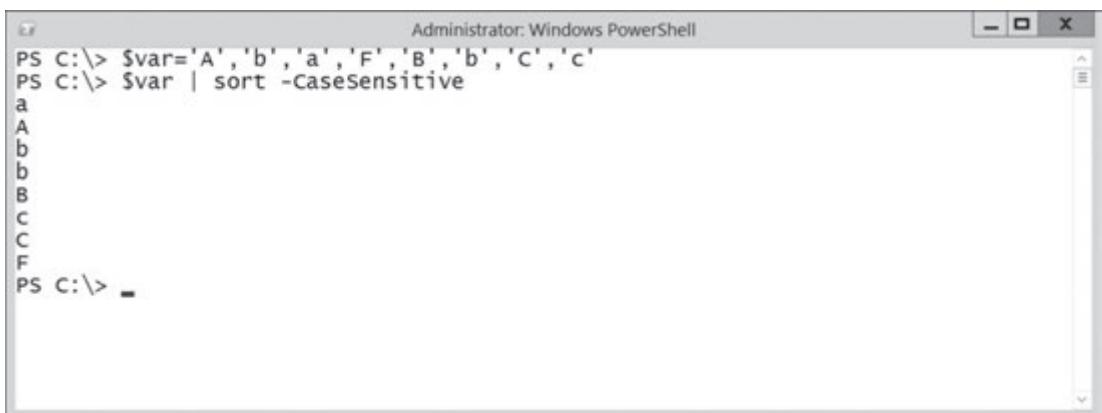
Figure 7-7

### Case Sensitivity

The **Sort-Object** cmdlet can also sort information by lower and upper case, by using the **-CaseSensitive** parameter.

```
$var = 'A', 'b', 'a', 'F', 'B', 'b', 'C', 'c'
```

```
$var | sort -CaseSensitive
```



Administrator: Windows PowerShell

```
PS C:\> $var='A','b','a','F','B','b','C','c'
PS C:\> $var | sort -CaseSensitive
a
A
b
b
B
C
C
F
PS C:\> _
```

Figure 7-8

You may not need this often, but it's available when you need it.

## Select and sort a subset of data

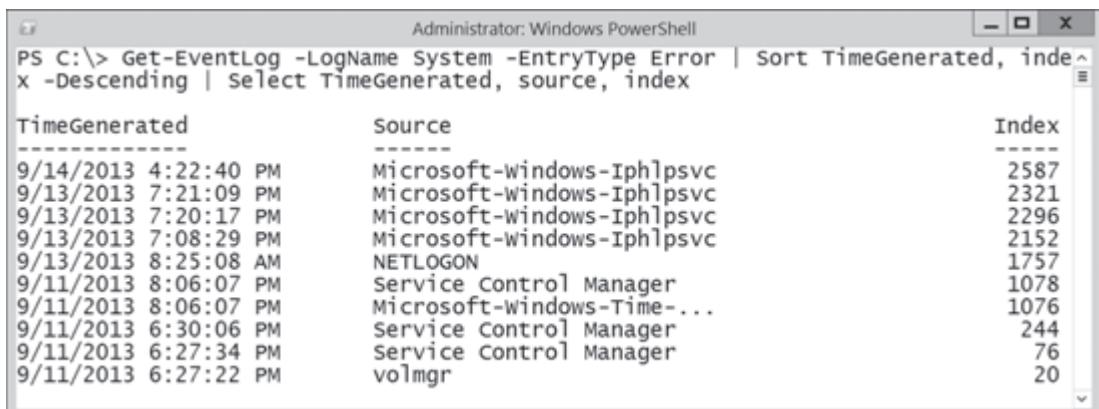
Combining **Sort** with **Select-Object** provides powerful sorting with the ability to select only the data you wish to see. You will use the two together often to generate reports and trim the results down to specific properties. Normally, it doesn't matter if you perform the sort or the select first in the pipeline, but as you will see, mistakes can happen. Let's start by combining the **Sort-Object** and **Select-Object** cmdlets.

---

### Note:

As a reminder, we will be switching between the full cmdlet and aliases throughout this book as your experience increases. It becomes useful to recognize commonly used aliases at the command console. Later, when we start scripting, we won't be using aliases in our scripts, for clarity and maintainability. However, they are used often at the command console.

```
Get-EventLog -LogName System -EntryType Error | Sort TimeGenerated, index -Descending | Select TimeGenerated, source, index
```



TimeGenerated	Source	Index
9/14/2013 4:22:40 PM	Microsoft-Windows-Iphlpsvc	2587
9/13/2013 7:21:09 PM	Microsoft-Windows-Iphlpsvc	2321
9/13/2013 7:20:17 PM	Microsoft-Windows-Iphlpsvc	2296
9/13/2013 7:08:29 PM	Microsoft-Windows-Iphlpsvc	2152
9/13/2013 8:25:08 AM	NETLOGON	1757
9/11/2013 8:06:07 PM	Service Control Manager	1078
9/11/2013 8:06:07 PM	Microsoft-Windows-Time-...	1076
9/11/2013 6:30:06 PM	Service Control Manager	244
9/11/2013 6:27:34 PM	Service Control Manager	76
9/11/2013 6:27:22 PM	volmgr	20

Figure 7-9

A common mistake that we see new PowerShell admins make is to try and sort on properties that don't exist. When you use **Select-Object**, it filters out the properties not selected—and attempting to sort on a missing property doesn't provide the correct results. Take the following example, which does not do what we want.

```
Get-service | Select name | Sort status
```

Sorting and measuring data for output

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-service | Select name | Sort status". The output lists service names in alphabetical order:

Name
SSDPSRV
sppsvc
svsvc
SstpSvc
Spooler
shellHWDetection
SharedAccess
SNMPTRAP
smphost
THREADORDER
Themes
TPAutoConnSvc

Figure 7-10

The problem is that the **Sort-Object** cmdlet is not working correctly and you may not even catch it! Here is a correct version—notice the difference in the results.

```
Get-service | Select name, status | Sort status
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-service | Select name, status | Sort status". The output displays service names and their current status, sorted by status (all services are listed as "Stopped").

Name	Status
NtFrs	Stopped
PerfHost	Stopped
pla	Stopped
NetTcpPortSharing	Stopped
VaultSvc	Stopped
upnphost	Stopped
TrustedInstaller	Stopped
RasAuto	Stopped
RasMan	Stopped
UmRdpService	Stopped
UIODetect	Stopped
PrintNotify	Stopped

Figure 7-11

This time, we provided the **Status** property so that the **Sort-Object** cmdlet would work properly. To avoid this confusion, we like to sort our results before selecting the properties to display. Perhaps we wanted to only display the names, but they were sorted by status.

```
Get-Service | Sort -Property status | Select -Property name
```

```
Administrator: Windows PowerShell
PS C:\> Get-Service | Sort -Property status | Select -Property name
Name
----
NtFrs
PerfHost
pla
NetTcpPortSharing
VaultSvc
upnhost
TrustedInstaller
RasAuto
RasMan
UmRdpService
UIODetect
PrintNotify
```

Figure 7-12

A good rule of thumb is to sort first, to avoid any confusion. However, it really doesn't matter, unless you attempt to sort on a property that is not selected for display.

## Measuring the results

Another useful cmdlet is **Measure-Object**. This cmdlet can count objects in the pipeline; calculate the minimum, maximum, sum, and average of integers, even count words and characters for strings.

Want to know how many services or processes are running on your computer? Don't count them from a list—make PowerShell do the work for you!

```
Get-service | Measure-Object
```

```
Administrator: Windows PowerShell
PS C:\> Get-service | Measure-Object
Count      : 149
Average    :
Sum        :
Maximum   :
Minimum   :
Property  :
```

```
PS C:\>
```

Figure 7-13

**Measure-Object** gives a nice simple count of objects, but it has much more flexibility than just counting. Suppose you wanted to know the total amount of virtual memory being used by all processes, including the smallest and largest amount. **Measure-Object** has a **-Property** parameter—so select the property that you want to measure, and then specify the type of results.

```
Get-Process | Measure-Object -Property VirtualMemorySize -Minimum -Maximum -Sum
```

```
Administrator: Windows PowerShell
PS C:\> Get-Process | Measure-Object -Property virtualMemorySize -Minimum -Maximum -Sum

Count      : 39
Average    :
Sum        : 7558348800
Maximum    : 1333956608
Minimum    : 65536
Property   : VirtualMemorySize

PS C:\>
```

Figure 7-14

Here is another example, using the **Get-ChildItem** cmdlet to retrieve a list of files from the C:\Windows directory.

```
get-childitem -Path c:\Windows -File | Measure-Object -Property length -Sum -Average -Maximum -Minimum
```

```
Administrator: Windows PowerShell
PS C:\> get-childitem -Path c:\Windows -File | Measure-Object -Property length -Sum -Average -Maximum -Minimum

Count      : 20
Average    : 206523.55
Sum        : 4130471
Maximum    : 2328880
Minimum    : 0
Property   : Length

PS C:\>
```

Figure 7-15

Sorting, selecting, and measuring results are powerful report and troubleshooting tools that allow you to analyze exactly the information you need. Well, perhaps not exactly the information you need, because we haven't shown you how to filter (or discard) objects from the pipeline. Try the exercise, and then dive into filtering your output to refine your results even further.

## Exercise 7 – Sorting and measuring data

**Time:** 20 minutes

Combining the skills of selecting and sorting the data you want is an important step working with PowerShell. Try these tasks then spend a few extra minutes going beyond this exercise and work with your own environment to produce some useful results.

### Task 1

Display a list of the 20 newest errors in the system log; sort by date and index. Only display the time, the index, and the message

### Task 2

Retrieve a list of your current processes. Sort the list so that the highest CPU is listed first, and then display only the first three processes.

### Task 3

Retrieve a list of volumes. Sort the list so that the drive letters appear alphabetically.

### Task 4

Display a list of hotfixes, sorted so that only the most recent ones appear first. Only display the HotfixID, the installation date, and who installed the hotfix.



## Chapter 8

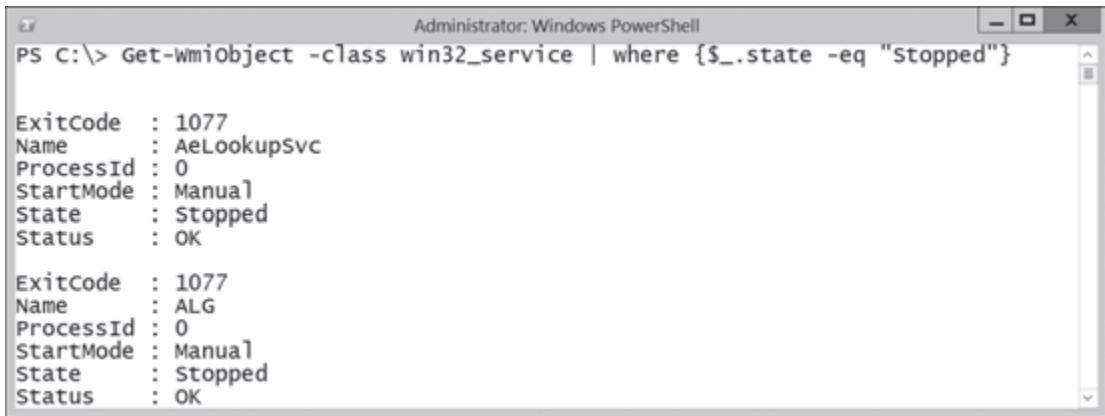
# Filtering your output

### You should filter your data

In addition to sorting, you may need to limit or filter the output. The **Where-Object** cmdlet is a filter that lets you control what data is ultimately displayed. This cmdlet is almost always used in a pipeline expression where output from one cmdlet is piped to this cmdlet. The **Where-Object** cmdlet requires a code block enclosed in braces and that code block is executed as the filter. Any input objects that match your criteria are passed down the pipeline; any objects that don't match your criteria are dropped.

Here's an expression to find all instances of the Win32\_Service class, where the **State** property of each object equals **Stopped**.

```
Get-WmiObject -class win32_service | where { $_.state -eq "Stopped" }
```



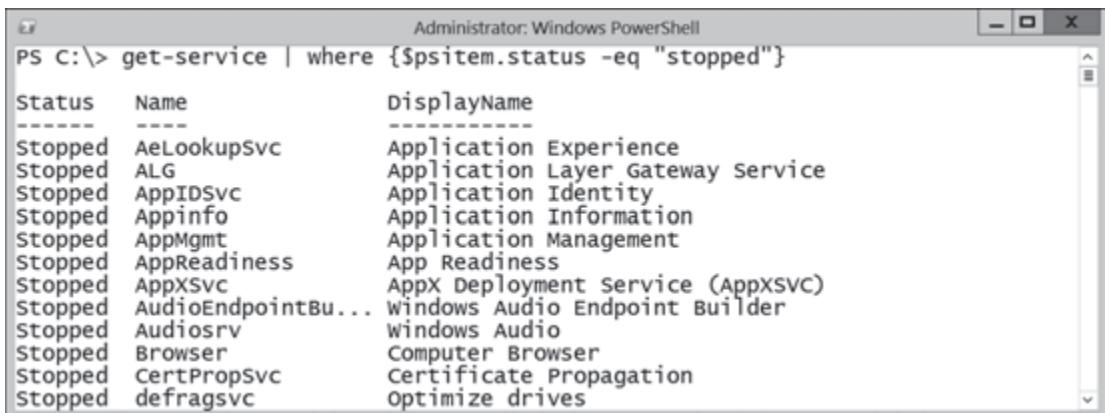
```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject -class win32_service | where {$_.state -eq "Stopped"}
```

ExitCode	Name	ProcessId	StartMode	Status
1077	AeLookupSvc	0	Manual	Stopped
1077	ALG	0	Manual	Stopped

Figure 8-1

In Figure 8-1, notice the use of the special `$_` variable, which represents the current pipeline object. So, that expression reads, “where the current object’s **State** property is equal to the value **Stopped**.” Starting with PowerShell version 3, a new variable is included that also represents the current object on the pipeline, **\$psitem**. You should become used to seeing and using both, as in the Figure 8-2 example of retrieving the stopped services by using **Get-Service**.

```
get-service | where { $psitem.status -eq "stopped" }
```



```
Administrator: Windows PowerShell
PS C:\> get-service | where {$psitem.status -eq "stopped"}
```

Status	Name	DisplayName
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Stopped	AppIDSvc	Application Identity
Stopped	Appinfo	Application Information
Stopped	AppMgmt	Application Management
Stopped	AppReadiness	App Readiness
Stopped	AppXsvc	Appx Deployment Service (Appxsvc)
Stopped	AudioEndpointBu...	Windows Audio Endpoint Builder
Stopped	Audiosrv	Windows Audio
Stopped	Browser	Computer Browser
Stopped	CertPropSvc	Certificate Propagation
Stopped	defragsvc	Optimize drives

Figure 8-2

The key is recognizing that the script block in braces is what filters the object. If nothing matches the filter, nothing will be displayed. Did you notice the strange looking **-eq**? Before we go any further with **Where-Object**, let’s talk about Comparison operators.

## Comparison operators

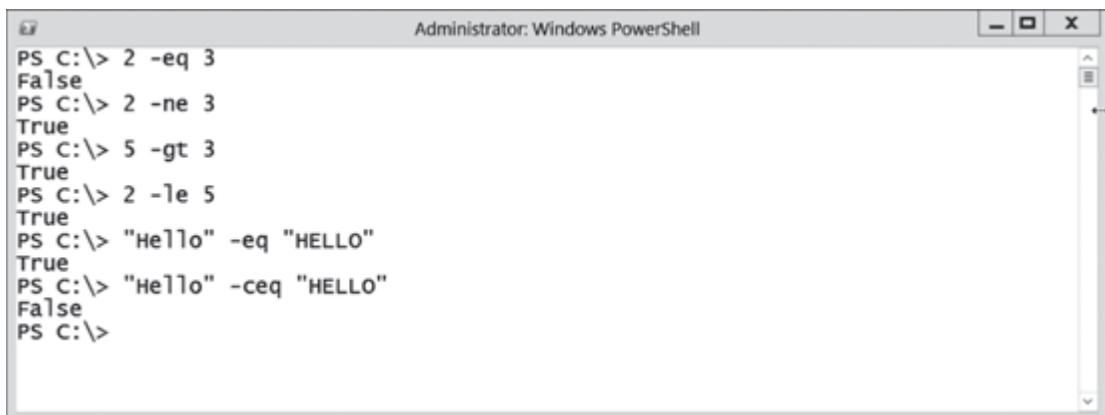
To filter out the objects that you don't want and keep the ones that you do, PowerShell needs you to define the criteria. If the criteria tests as **True**, PowerShell will keep the object; if the criteria tests as **False**, they are dropped. A common way to accomplish this is with an expression that uses comparison operators. These are not something you need to memorize—you can always check out the **Get-Help** for `about_Comparison_Operators`, but we want to show you a few of the more common ones.

### PowerShell Comparison Operators

<u>Operator</u>	<u>Description</u>	<u>Algebraic Equivalent</u>
<code>-eq</code>	Equals	$A=B$
<code>-ne</code>	Not equal	$A \neq B$
<code>-gt</code>	Greater than	$A > B$
<code>-ge</code>	Greater than or equal to	$A \geq B$
<code>-lt</code>	Less than	$A < B$
<code>-le</code>	Less than or equal to	$A \leq B$

Here are some simple examples using comparison operators. Notice that when the comparison is made, **True** or **False** is returned. When you use this with a cmdlet like **Where-Object**, **True** results are sent down the pipeline and **False** results are thrown out.

These operators are all case insensitive, as far as PowerShell is concerned. Case sensitive operators (`-ceq`, `-cne`, `-cgt`, `-cge`, `-clt`, and `-cle`) are available to compare strings.



```
Administrator: Windows PowerShell
PS C:\> 2 -eq 3
False
PS C:\> 2 -ne 3
True
PS C:\> 5 -gt 3
True
PS C:\> 2 -le 5
True
PS C:\> "Hello" -eq "HELLO"
True
PS C:\> "Hello" -ceq "HELLO"
False
PS C:\>
```

Figure 8-3

These comparison operators work just fine for simple numeric comparisons. For string comparisons, we can use **-Like** and **-Match**. If your comparison needs are simple, the **-Like** operator may be all you need.

```
"10.10.10.1" -like "10.10.10.*"
```

```
"10.10.10.25" -like "10.10.10.*"
```

```
"10.10.11.1" -like "10.10.10.*"
```

The screenshot shows an Administrator Windows PowerShell window with the title bar 'Administrator: Windows PowerShell'. The command PS C:\> "10.10.10.1" -like "10.10.10.\*" is run, followed by PS C:\> "10.10.10.25" -like "10.10.10.\*", and finally PS C:\> "10.10.11.1" -like "10.10.10.\*". The output for the first two commands is 'True', and for the third is 'False'.

```
Administrator: Windows PowerShell
PS C:\> "10.10.10.1" -like "10.10.10.*"
True
PS C:\> "10.10.10.25" -like "10.10.10.*"
True
PS C:\> "10.10.11.1" -like "10.10.10.*"
False
PS C:\>
```

Figure 8-4

---

**Note:**

The **-Like** and **-Match** operators are also case insensitive. The **-Clike** and **-Cmatch** operators are their case sensitive counterparts.

In Figure 8-4, we're comparing an IP address (10.10.10.1) to a pattern that uses the wildcard character (\*). This comparison returns the Boolean value **True**. The second comparison also matches, but the third fails. The operator returns **False** because the third octet no longer matches the pattern.

Depending on the logic of your script, you may want to check the inverse. In other words, you may want to determine whether the address is not like the pattern. For an inverse type of comparison, use the **-NotLike** operator.

```
"10.10.11.1" -notlike "10.10.10.*"
```

This is essentially the same comparison, except this operator returns **True** because 10.10.11.1 is not like 10.10.10.\*.

The asterisk is a multiple character wildcard that can be used if we need something more granular. For example, we can use the ? operator if we want to match any subnet of 10.10.10.x to 10.10.19.x.

```
"10.10.11.1" -like "10.10.1?.*"
```

```
"10.10.15.1" -like "10.10.1?.*"
```

```
"10.10.25.1" -like "10.10.1?.*"
```

In this example, you can see that the first two comparisons are **True**; but the last one does not meet the pattern, so it returns **False**.

## Text comparison only

Make sure you understand that the IP address comparisons are merely looking at each IP address string. We are not calculating or comparing network address with subnet masks or the like.

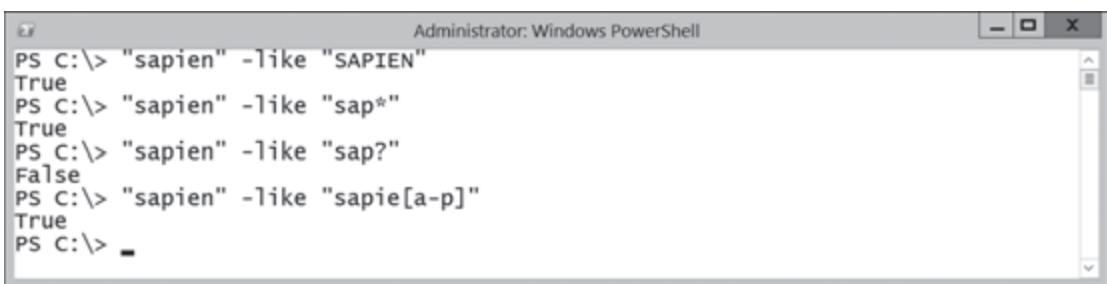
Let's look at some text comparison examples.

```
"sapien" -like "SAPIEN"
```

```
"sapien" -like "sap*"
```

```
"sapien" -like "sap?"
```

```
"sapien" -like "sapie[a-p]"
```



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command 'PS C:\> "sapien" -like "SAPIEN"' returns 'True'. The command 'PS C:\> "sapien" -like "sap\*' returns 'True'. The command 'PS C:\> "sapien" -like "sap?'" returns 'False'. The command 'PS C:\> "sapien" -like "sapie[a-p]"' returns 'True'. The prompt 'PS C:\>' is visible at the bottom.

```
Administrator: Windows PowerShell
PS C:\> "sapien" -like "SAPIEN"
True
PS C:\> "sapien" -like "sap*"
True
PS C:\> "sapien" -like "sap?"
False
PS C:\> "sapien" -like "sapie[a-p]"
True
PS C:\> _
```

Figure 8-5

The first example is a pretty basic comparison that also demonstrates that the **-Like** operator is case insensitive. The second example uses the wildcard character, which means it will return **True** for a string like “sapien”. However, it will also return **True** for “sapsucker” and “sapling”. The third example returns **False** because the “?” character means any single character after “sap”. The last example is a bit different. We can use brackets to denote a range of characters with which to compare. In this last case, the **-Like** operator will return **True** for anything that starts with “sapi” and ends with any character between “a” and “p”.

The **-Like** operator limits your comparisons essentially to a few wildcards. If you need something that will compare a pattern, use the **-Match** operator. This operator also returns **True** if the string matches the specified pattern.

At its simplest, **-Match** returns **True** if any part of the string matches the pattern.

```
"Win8Desktop" -match "Win8"
```

```
"Win8Desktop" -match "Win"
```

```
"Win8Desktop" -match "in"
```

```
"Win8Desktop" -match "fun"
```

```
Administrator: Windows PowerShell
PS C:\> "Win8Desktop" -match "Win8"
True
PS C:\> "Win8Desktop" -match "Win"
True
PS C:\> "Win8Desktop" -match "in"
True
PS C:\> "Win8Desktop" -match "fun"
False
PS C:\>
```

Figure 8-6

The **-Match** operator is very powerful and supports regular expressions, which are discussed later in the book. Before we return to using the comparison operators in a filter, let's make a quick stop at logical operators.

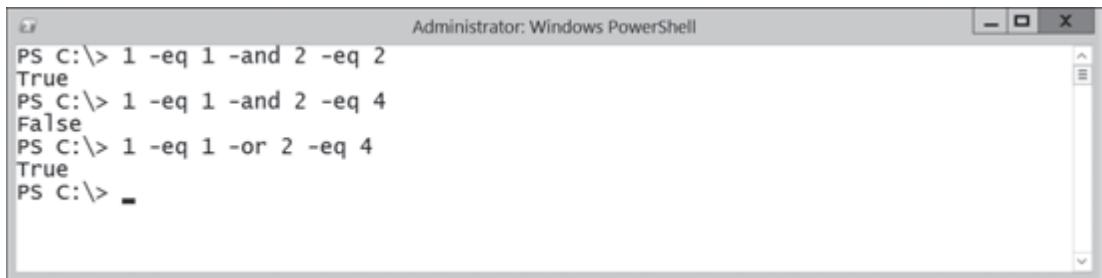
## Logical operators

In PowerShell, logical operators are used to test or validate an expression. Typically, the result of these operations is **True** or **False**. Here is a list of the operators.

### PowerShell Logical Operators

<u>Operator</u>	<u>Description</u>	<u>Example</u>
-and	All expressions must evaluate as TRUE.	(1 -eq 1) -and (2 -eq 2) returns TRUE
-or	At least one expression must evaluate as TRUE.	(1 -eq 1) -or (2 -eq 4) returns TRUE
-not	Evaluates the inverse of one of the expressions.	(1 -eq 1) -and -not (2 -gt 2) returns TRUE
!	The same as -not.	(1 -eq 1) -and !(2 -gt 2) returns TRUE

You should use logical operators when you want to evaluate multiple conditions. While you can evaluate several expressions, your scripts and code will be easier to debug or troubleshoot if you limit the operation to two expressions.



```
Administrator: Windows PowerShell
PS C:\> 1 -eq 1 -and 2 -eq 2
True
PS C:\> 1 -eq 1 -and 2 -eq 4
False
PS C:\> 1 -eq 1 -or 2 -eq 4
True
PS C:\> _
```

Figure 8-7

Using comparison and logical operators, let's go back to working with **Where-Object**.

## More on Where-Object

Now that you are armed with comparison and logical operators, let's run through a few more examples of filtering with **Where-Object**.

```
Get-Process | Where-Object { $PSItem.CPU -gt 1 }
```

```
Administrator: Windows PowerShell
PS C:\> Get-Process | Where-Object { $PSItem.CPU -gt 1 }

Handles  NPM(K)    PM(K)      WS(K)  VM(M)   CPU(s)    Id  ProcessName
----  -----  -----  -----  -----  -----  --  -----
  907     49    17660    47340   299    1.09    2752  explorer
1268    151    55444    49760   1272    1.73    512  lsass
  476     26    73372    77708   620    1.23    2040 powershell
1205     51    14484    28488   131    1.33    852  svchost
  699     0     112      328     3    4.61     4  System
  318     26    10320    20468   149    3.58    1832 vmtoolsd

PS C:\>
```

Figure 8-8

In the expression shown in Figure 8-8, we are using **\$PSItem**, which holds the current object in the pipeline. Using dot-notation, we are selecting a property of the object, in this case the CPU(s), and looking for any process that is greater than 1 CPU(s).

In the following example, we only want processes greater than 1 CPU(s) and that also have a name that begins with "p".

```
Get-Process | Where-Object { $PSItem.CPU -gt 1 -and $PSItem.name -like "p*" }
```

```
Administrator: Windows PowerShell
PS C:\> Get-Process | Where-Object { $PSItem.CPU -gt 1 -and $PSItem.name -like "p*"}

Handles  NPM(K)    PM(K)      WS(K)  VM(M)   CPU(s)    Id  ProcessName
----  -----  -----  -----  -----  -----  --  -----
  499     26    72368    77932   620    1.39    2040 powershell

PS C:\>
```

Figure 8-9

In Figure 8-10, we filter only those services that have a name that begins with a “b” and have a status of **Running**.

```
Get-Service | Where-Object { $_.Name -like "b*" -and $_.status -eq "Running" }
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-Service | Where-Object { \$\_.Name -like "b\*" -and \$\_.status -eq "Running" }". The output displays three services: BFE, BITS, and BrokerInfrastructure, all in a "Running" status. The output is formatted as a table with columns for Status, Name, and DisplayName.

Status	Name	DisplayName
Running	BFE	Base Filtering Engine
Running	BITS	Background Intelligent Transfer Ser...
Running	BrokerInfrastructure	Background Tasks Infrastructure Ser...

Figure 8-10

---

**Note:**

Remember that \$PSCmdlet could be replaced with \$\_—you will see both.

These are simple examples and they are a good starting point. Before you try this in the exercise, there are a few more rules that we need to discuss.

## Best performance for filtering

We see many admins make a few mistakes—and you will want to avoid them. Take a look at the following two examples and consider what is happening as the object moves down the pipeline. The result is the same, but there is something very different happening under the hood.

```
Get-Service | Sort-Object -Property status | Where { $_.name -like "b*" }
```

```
Get-Service | Where { $_.name -like "b*" } | sort-object -property status
```

```

Administrator: Windows PowerShell
PS C:\> Get-Service | Sort-Object -Property status | Where {$_.name -like "b*"}
PS C:\> Get-Service | Where {$_.name -like "b*"} | sort status

Status      Name          DisplayName
-----      --           -----
Stopped    Browser       Computer Browser
Running    BFE           Base Filtering Engine
Running    BrokerInfrastru... Background Tasks Infrastructure Ser...
Running    BITS          Background Intelligent Transfer Ser...

PS C:\>

```

Figure 8-11

In the first example, we are gathering all the service objects, sorting ALL of them by status, then filter out only the ones that have a service name that begins with “b\*”. Do you see the problem? We wasted time sorting data we didn’t even want!

The next example filters out only the objects that we want, and then performs the sort. This is much better, because we are not wasting time sorting information that we don’t want. While this simple example doesn’t perform differently, when you start working with larger amounts of data, this will make a big difference in performance. We always try to filter as far left as possible in the pipeline—and if available—filtering with parameters on the first cmdlet. This will reduce the amount of objects that we are sending down the pipeline and improve the performance when working with large amounts of objects.

In fact, many cmdlets have parameters that perform filter operations. The reason? So we don’t waste time on objects that we don’t want. When you are examining the Help file for a cmdlet, try to notice the parameters that act as filters. As an example, the **Get-Service** cmdlet has a **-Name** parameter. In other words, in the example shown in Figure 8-11, we don’t need the **Where-Object** cmdlet.

```
Get-Service -name b* | Sort -Property status
```

```

Administrator: Windows PowerShell
PS C:\> Get-Service -name b* | Sort -Property status

Status      Name          DisplayName
-----      --           -----
Stopped    Browser       Computer Browser
Running    BFE           Base Filtering Engine
Running    BrokerInfrastru... Background Tasks Infrastructure Ser...
Running    BITS          Background Intelligent Transfer Ser...

PS C:\>

```

Figure 8-12

The goal is to filter as far left as possible, so start by examining parameters that might do the job. If there are no parameters or you still need to filter further than the cmdlet supports, pipe to **Where-Object**. Perform sorting and other operations after the filtering is complete.

## Exercise 8 – Filtering your output

**Time: 20 minutes**

The goal to this exercise is to filter out the results you wish to see. Remember: filter as far left as possible.

### Task 1

Display a list of all servers that begin with the letter “b” and that are currently stopped.

### Task 2

Display a list of all volumes that have less than 10 percent of free disk space remaining. (You may need to change this number to have a result—we don’t know how much disk space you have on your computer.)

### Task 3

Display a list of all running process with a CPU greater than 1, sort them so the highest CPU appears first, and then only display the name of the process and its CPU.

### Task 4

Display a list of all hotfixes that are security updates.

### Task 5

Display a list of only the \*.PS1 files on your system. Check the entire drive.



## Chapter 9

# Exporting and formatting your data

### Useful reporting and more!

PowerShell's ability to manipulate objects is pretty formidable. Gathering, selecting, sorting, and filtering only the information we want is important to analyzing and resolving problems. Often, you will want to store this information for later review or to print reports for other people.

PowerShell also brings a lot of flexibility that at first might create some confusion. One day you might want a CSV of your service information, the next day you might want it stored in XML, or as a web report on a web server.

To accommodate all the variety of needs, PowerShell has several types of cmdlets to help you out. You can find detailed Help information on the **Export**, **Import**, and **Format** cmdlets discussed in this chapter—and you should—as your needs will change from day-to-day.

### Exporting and importing comma-separated value (CSV) files

A comma-separated value (CSV) file is a mainstay of administrative scripting. It's a text-based database that you can parse into an array or open in a spreadsheet program, like Microsoft Excel. The **Export-Csv** cmdlet requires an input object that is typically the result of a piped cmdlet.

```
Get-Process | Export-Csv -Path c:\Process.csv
```

When you run this command on your system, it creates a text file called Processes.csv. If you want to retrieve the information from the file—or any CSV—use the **Import-CSV** cmdlet.

```
Get-Process | Export-Csv -Path c:\Process.csv
```

```
Import-Csv -Path C:\Process.csv
```

__NounName	:	Process
Name	:	conhost
Handles	:	56
VM	:	55975936
WS	:	7233536
PM	:	1732608
NPM	:	7408
Path	:	C:\windows\system32\conhost.exe
Company	:	Microsoft Corporation
CPU	:	0.671875
FileVersion	:	6.3.9600.16384 (winblue_rtm.130821-1623)

Figure 9-1

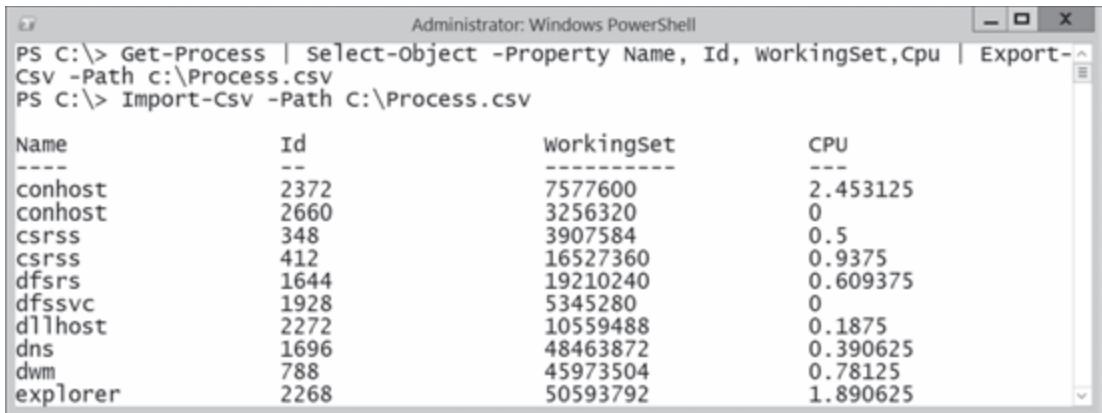
You might be wondering at this point why the displayed data doesn't match what you normally see when you run **Get-Process**. The **Export-CSV** cmdlet stored all the properties to the file, not just the ones you see in the default view. This is probably overkill most of the time, so you will want to use the skills that you learned from previous chapters to select and filter only the data that you want.

Below is another version of the expression in Figure 9-1, except this time we're using **Select-Object** to specify the properties that we want returned.

```
Get-Process | Select-Object -Property Name, Id, WorkingSet, Cpu | Export-Csv -Path c:\Process.csv
```

```
Import-Csv -Path C:\Process.csv
```

## Exporting and formatting your data



```
Administrator: Windows PowerShell
PS C:\> Get-Process | Select-Object -Property Name, Id, WorkingSet,Cpu | Export-Csv -Path c:\Process.csv
PS C:\> Import-Csv -Path C:\Process.csv

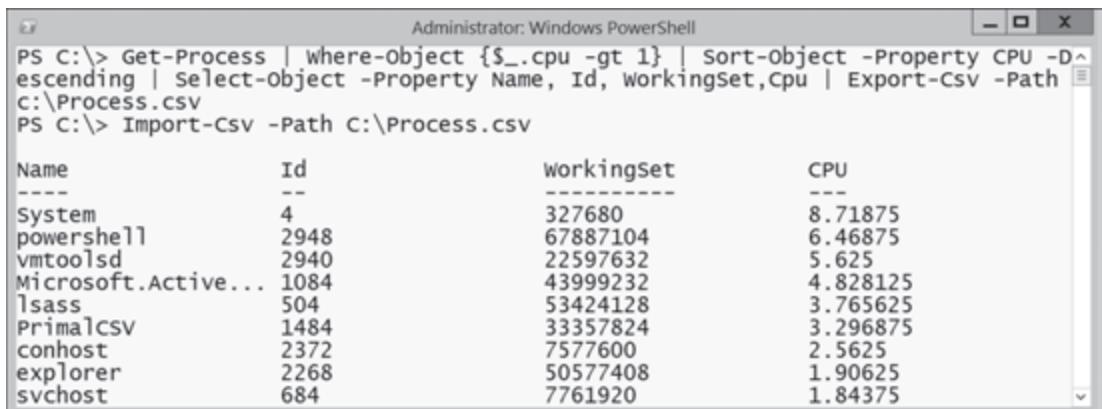
Name          Id      WorkingSet      CPU
---          --      -----      ---
conhost       2372    7577600     2.453125
conhost       2660    3256320     0
csrss         348     3907584     0.5
csrss         412     16527360    0.9375
dfssrs        1644    19210240    0.609375
dfssvc        1928    5345280     0
dllhost       2272    10559488    0.1875
dns           1696    48463872    0.390625
dwm           788     45973504    0.78125
explorer      2268    50593792    1.890625
```

Figure 9-2

This may still be too much data, so keep building the command to fit your needs, then export. In this case, we want a CSV file with only the processes that have a CPU greater than 1.

```
Get-Process | Where-Object { $_.cpu -gt 1 } | Sort-Object -Property CPU -Descending | Select-Object -Property Name, Id, WorkingSet, Cpu | Export-Csv -Path c:\Process.csv
```

```
Import-Csv -Path C:\Process.csv
```



```
Administrator: Windows PowerShell
PS C:\> Get-Process | where-Object {$_._cpu -gt 1} | Sort-Object -Property CPU -Descending | Select-Object -Property Name, Id, WorkingSet,Cpu | Export-Csv -Path c:\Process.csv
PS C:\> Import-Csv -Path C:\Process.csv

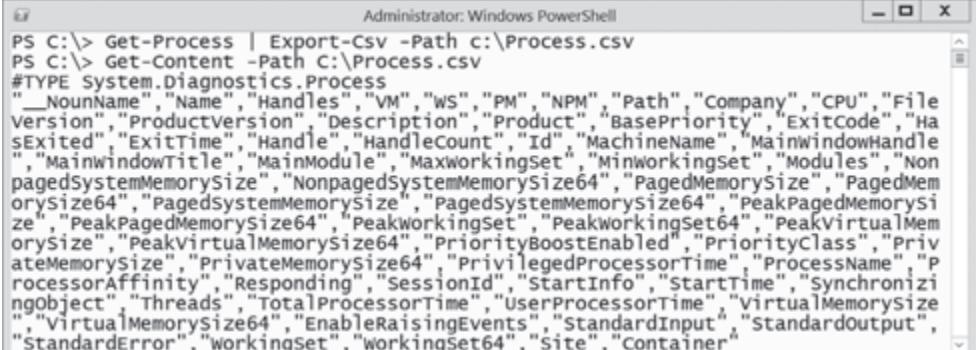
Name          Id      WorkingSet      CPU
---          --      -----      ---
System        4       327680      8.71875
powershell    2948   67887104    6.46875
vmtoolsd     2940   22597632    5.625
Microsoft.Active... 1084  43999232    4.828125
lsass         504    53424128    3.765625
PrimalCSV    1484   33357824    3.296875
conhost       2372    7577600     2.5625
explorer      2268   50577408    1.90625
svchost       684    7761920     1.84375
```

Figure 9-3

Yes, this command is getting a bit lengthy, but it only took a second to type, by using tab completion. We filtered the results so that only processes with a CPU greater than 1 will be saved in the CSV. We also wanted to sort the processes with the highest CPU listed first, then selected only the properties we care about. After we achieved the results we wanted, we exported the results to a CSV. PowerShell's ability to manipulate and convert objects to different forms of data allows you to create very useful reports.

## Opening the CSV in your favorite application

Before you attempt to open the CSV in a spreadsheet application, such as Microsoft Excel, you need to become familiar with the parameter **-NoTypeInformation**. By default, the object (remember **Get-Member**?) is written to the first line of the CSV file. Notice in Figure 9-4 that instead of importing the CSV, we displayed it with **Get-Content**.

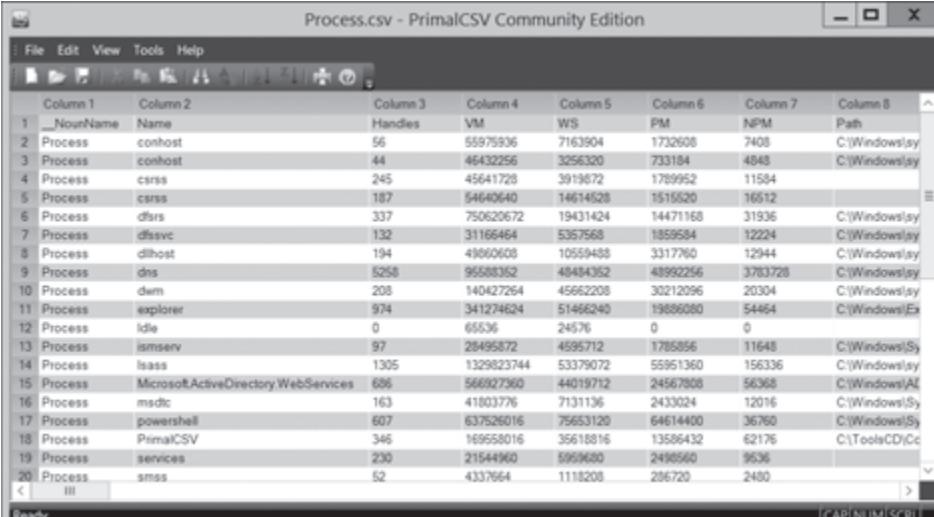


```
Administrator: Windows PowerShell
PS C:\> Get-Process | Export-Csv -Path c:\Process.csv
PS C:\> Get-Content -Path C:\Process.csv
#TYPE System.Diagnostics.Process
  "__NounName", "Name", "Handles", "VM", "WS", "PM", "NPM", "Path", "Company", "CPU", "FileVersion",
  "ProductVersion", "Description", "Product", "BasePriority", "ExitCode", "HasExited",
  "ExitTime", "Handle", "HandleCount", "Id", "MachineName", "MainWindowHandle",
  "MainWindowTitle", "MainModule", "MaxWorkingSet", "MinWorkingSet", "Modules", "Non
  pagedSystemMemorySize", "NonpagedSystemMemorySize64", "PagedMemorySize", "PagedMem
  orySize64", "PagedSystemMemorySize", "PagedSystemMemorySize64", "PeakPagedMemorySi
  ze", "PeakPagedMemorySize64", "PeakWorkingSet", "PeakWorkingSet64", "PeakVirtualMem
  orySize", "PeakVirtualMemorySize64", "PriorityBoostEnabled", "PriorityClass", "Pri
  vateMemorySize", "PrivateMemorySize64", "PrivilegedProcessorTime", "ProcessName", "P
  rocessorAffinity", "Responding", "SessionId", "StartInfo", "StartTime", "Synchroniz
  eObject", "Threads", "TotalProcessorTime", "UserProcessorTime", "VirtualMemorySize",
  "VirtualMemorySize64", "EnableRaisingEvents", "StandardInput", "StandardOutput",
  "StandardError", "WorkingSet", "WorkingSet64", "Site", "Container"
```

Figure 9-4

Notice that the first line of the file starts with **#TYPE**. This confuses many third-party CSV readers and needs to be removed. No problem! The **Export-CSV** cmdlet is designed for this—just add the parameter **-NoTypeInformation**.

```
Get-Process | Export-Csv -Path c:\Process.csv -NoTypeInformation
```



Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8
1	__NounName	Handles	VM	WS	PM	NPM	Path
2	Process	conhost	56	55975936	7163904	1732608	7408
3	Process	conhost	44	46432256	3256320	731384	4848
4	Process	csrss	245	45641728	3919872	1789952	11584
5	Process	csrss	187	54640640	14614528	1515520	16512
6	Process	dfrs	337	750620672	19431424	14471168	31936
7	Process	dfsvc	132	31166464	5357568	1859584	12224
8	Process	dihost	194	49860600	10594688	3317760	12944
9	Process	dns	5258	95588352	49484352	43992256	3783728
10	Process	dwm	208	140427264	45662208	30212096	20304
11	Process	explorer	974	341274624	51466240	19886080	54464
12	Process	idle	0	65536	24576	0	0
13	Process	ismserv	97	28495872	4595712	1765056	11648
14	Process	lsass	1305	1329823744	53379072	55951360	156336
15	Process	Microsoft.ActiveDirectory.WebServices	686	566927360	44019712	24567808	56368
16	Process	msdtc	163	41803776	7131136	2433024	12016
17	Process	powershell	607	637526016	79653120	64614400	36760
18	Process	PrimalCSV	346	169558016	35618816	13586432	62176
19	Process	services	230	21544960	5996800	2498560	9636
20	Process	sms	52	4337664	1118208	286720	2480

Figure 9-5

## Exporting and formatting your data

Your results will then be correctly displayed in your spreadsheet application or CSV reporting tool. We like using SAPIEN's PrimalCSV to work with CSV files.

### More about CSV files

By the way, in actuality **Import-CSV** reads a CSV file and assumes that the first line consists of column names. It then creates a collection of objects based upon the contents of the CSV file. For example, you can create your own CSV file in a simple text editor that **Import-CSV** can read.

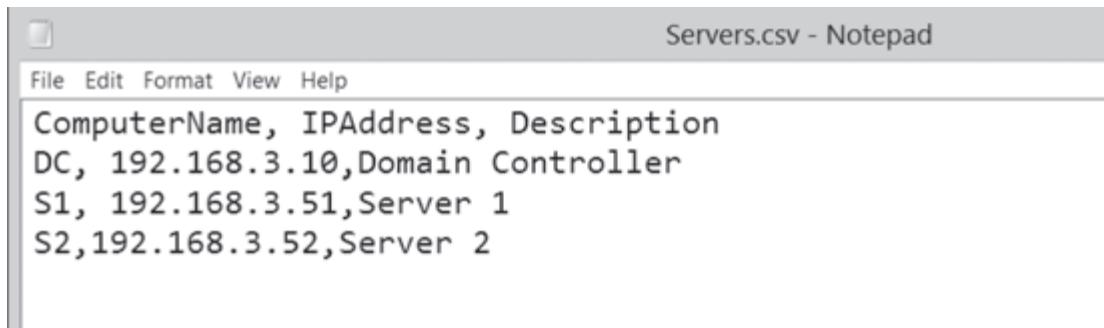


Figure 9-6

We created a simple server list with IP addresses and descriptions, by using Notepad. When you import this using **Import-CSV**, three objects are created, each with a **ComputerName**, **IP Address**, and **Description** property.

```
import-csv C:\Servers.csv
```

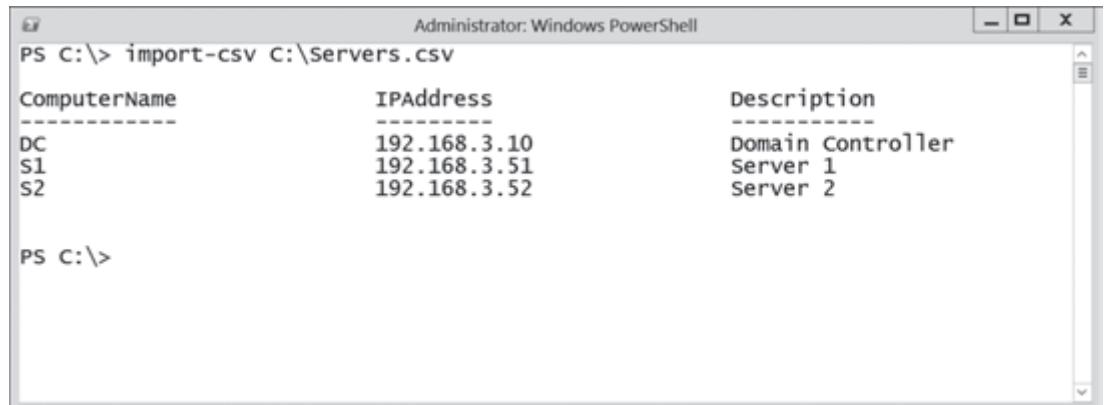


Figure 9-7

You can prove that these are objects with properties by piping the results to **Get-Member**.

```
import-csv C:\Servers.csv | gm
```

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
ComputerName	NoteProperty	System.String ComputerName=DC
Description	NoteProperty	System.String Description=Domain Controller
IPAddress	NoteProperty	System.String IPAddress=192.168.3.10

Figure 9-8

## Import- and Export- other delimited data

There is much you can learn from the Help files on **Export-CSV** and **Import-CSV**, and we will finish with one other common parameter. Not all CSV files are comma delimited—as the name implies—in fact, it is common to require other delimiters such as “;” or “|”. Both **Import-CSV** and **Export-CSV** support a **-Delimiter** parameter, which allows you to specify a data delimiter other than a comma. By using this parameter, you can import and export tab-delimited files (use `t as the delimiter), space-delimited files, pipe-delimited files, or whatever you might need.

## Exporting and importing XML

Now that you have the concept of exporting and importing your data for reports, there will be times when you want your data in Extensible Markup Language (XML). This is very handy if you want to transfer data to a database or other XML tools. We do this often.

### Export-CliXML

**Export-CliXML** works much the same way as **Export-CSV**. If you’re wondering, CliXML, stands for Command-Line Interface XML.

```
Get-Process | Select-Object -Property Name, Id, WorkingSet, Cpu | Export-Clixml -Path c:\Process.xml
```

```
Import-Clixml -Path C:\Process.xml
```

```

Administrator: Windows PowerShell
PS C:\> Get-Process | Select-Object -Property Name, Id, WorkingSet,Cpu | Export-Clixml -Path c:\Process.xml
PS C:\> Import-Clixml -Path C:\Process.xml

Name                Id      WorkingSet          CPU
----              --      -----          ---
conhost            2372    7581696        3.09375
conhost            2660    3256320          0
csrss              348     3907584        0.5
csrss              412     16728064       1.40625
dfsrs              1644    19406848       0.703125
dfssvc              1928    5345280          0
dllhost             2272    10559488       0.1875
dns                 1696    48508928       0.390625
dwm                  788     45244416       0.828125
explorer            2268    50651136       2.09375

```

Figure 9-9

This creates an XML file called Process.xml, which you can then import back into PowerShell by using **Import-CliXML**, very similar to using **Export/Import-CSV**. If you open the file in a text editor, you will discover the well-formed XML.

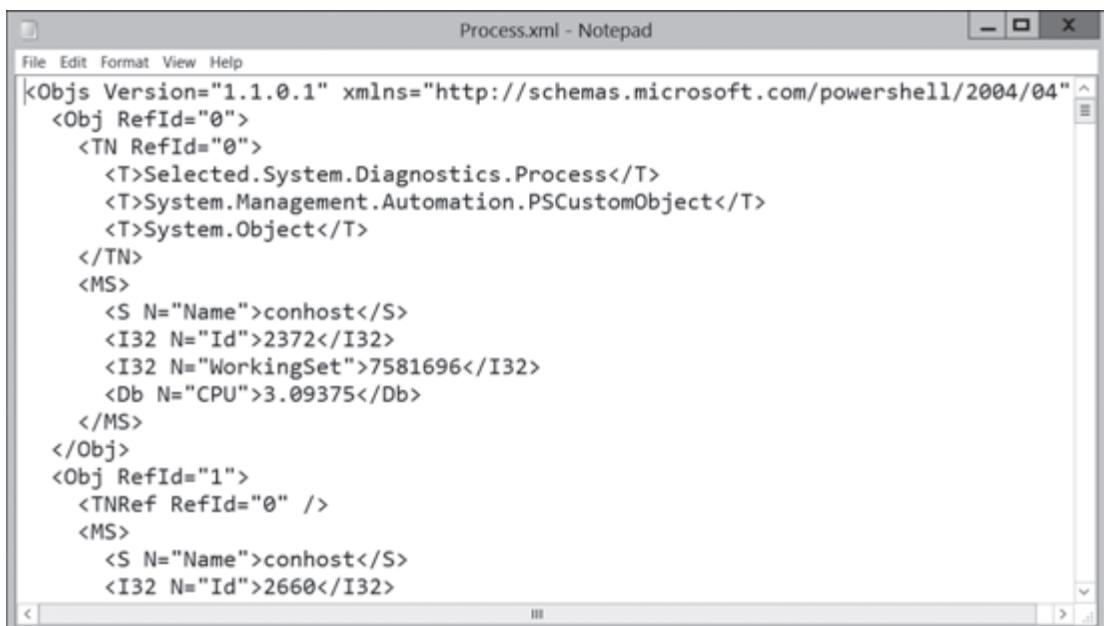


Figure 9-10

---

**Note:**

As with the other exporting cmdlets, you can use the parameter `-NoClobber` to avoid overwriting an existing file.

Exporting your information to XML can be a useful technique such as when used with command-line history. First, export it to an XML file.

```
Get-History | Export-Clixml c:\history.xml
```

Later, when you want to re-import that command-line history into the shell, use **Import-CliXML** and pipe the results to **Add-History**.

```
Import-Clixml c:\history.xml | Add-History
```

You can export most simple objects into the CliXML format, and then re-import them at a later date to approximately re-create those objects. The technique works best for objects that only have properties, where those properties only contain simple values such as strings, numbers, and Boolean values.

## Comparing objects and collections

Occasionally, you may need to compare complex objects or collections—with text files perhaps being the most common and easily explained example. PowerShell provides a **Compare-Object** cmdlet, which can perform object comparisons. Now, before we get started, we have to remind you: PowerShell deals in objects, not text. A text file is technically a collection of individual string objects; that is, each line of the file is a unique, independent object, and the text file serves to collect them together.

When comparing objects, you start with two sets: the reference set and the difference set. In the case of two text files, these would simply be the two files. Which one is the reference and which one is the difference often doesn't matter. **Compare-Object** will show you which objects—that is, which lines of text—exist in one set but not the other or which are present in both sets, but different. That last bit—showing you objects which are present in both sets, but different—can get confusing, so we'll talk about that later.

To begin, we're going to use two text files named `Text1.txt` and `Text2.txt`, with some simple lines of text in each file.

### Text1.txt

PowerShell Rocks!

PowerShell is fun!

## Text2.txt

```
PowerShell is great!
```

```
PowerShell is fun!
```

We'll use **Get-Content** in a parenthetical for the parameters to load each file and perform the comparison.

```
Compare-Object -ReferenceObject (Get-Content C:\Text1.txt) -DifferenceObject (Get-Content C:\Text2.txt)
```

InputObject	SideIndicator
PowerShell is great!	=>
PowerShell Rocks!	<=

Figure 9-11

The **SideIndicator** column tells us which side of the comparison was unequal. The reference set is always thought of as being on the left, while the difference set is thought of as being on the right. And remember, we're only seeing the differences here, so the text line "PowerShell is Fun!" will not be displayed because it's the same in both files.

## Real-world comparison example

Let's create a real-world example using XML and **Compare-Object**. We have two computers, one is running perfectly and the other is not. We suspect there are some processes running that might be causing a problem on the computer that is running slowly and we wish to compare the two computers to determine if this is the case.

On the well-running computer, we will start by gathering the process information and storing it into an XML file.

```
Get-Process | Export-Clixml -Path c:\GoodComputer.xml
```

We will take that file over to the computer running slowly and perform the comparison.

```
Compare-Object -ReferenceObject (Import-Clixml C:\GoodComputer.xml) -DifferenceObject (Get-Process) -Property Name
```

```
Administrator: Windows PowerShell
PS C:\> Compare-Object -ReferenceObject (Import-Clixml C:\GoodComputer.xml) -DifferenceObject (Get-Process) -Property Name
Name           SideIndicator
----           -----
calc           =>
mspaint        =>
notepad        =>

PS C:\>
```

Figure 9-12

In the command in Figure 9-12, we are using a parenthetical to import the good XML file, and then using that value for the **-ReferenceObject** parameter. For the **-DifferenceObject** parameter, again we are using a parenthetical, but this time we are running the **Get-Process** cmdlet for the value. Since **Get-Process** has many different properties, we need to tell the **Compare-Object** cmdlet which property we want it to focus its comparison. To accomplish this, we added the **-Property** parameter and specified the **Name** property.

From the results, you can tell that the computer running slowly (**-DifferenceObject**) is running some suspect processes that the well-running computer is not. Of course, in our example the applications Calc, MSPaint, and Notepad would never create this kind of problem—this is just an example.

## Converting to CSV and XML

As you are working with PowerShell, at some point you will stumble across the **ConvertTo-** cmdlets. At first glance, you will probably wonder why **ConvertTo-CSV** and **ConvertTo-xml** exist. It may seem strange, because you already have the **Export-** and **Import-** cmdlets for these.

The **ConvertTo-** cmdlets perform almost the same task as the **Export-** cmdlets. In fact, the data file will be very similar, with one critical difference. The **ConvertTo-** cmdlets leave the object in the pipeline, whereas the **Export-** cmdlets do not.

```
Get-Process | Export-Csv -Path c:\Process.csv | gm
```

```

Administrator: Windows PowerShell
PS C:\> Get-Process | Export-Csv -Path c:\Process.csv | gm
gm : You must specify an object for the Get-Member cmdlet.
At line:1 char:49
+ Get-Process | Export-Csv -Path c:\Process.csv | gm
+ ~~~~~
+ CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperationException
+ FullyQualifiedErrorMessage : NoObjectInGetMember,Microsoft.PowerShell.Commands.GetMemberCommand
+ S
PS C:\>

```

Figure 9-13

Notice that we used **Export-CSV**—just like earlier—but this time we piped it to **Get-Member**. The error message tells us that “You must specify an object for the **Get-Member** cmdlet.” In other words, when you use **Export-** cmdlets, they must be the last command in the pipeline. In effect, you’re done. The **Convert-** cmdlets leave the objects in the pipeline so that you can continue to work with them.

```
Get-Process | ConvertTo-Csv | gm
```

```

Administrator: Windows PowerShell
PS C:\> Get-Process | ConvertTo-Csv | gm
TypeName: System.String
Name      MemberType
----      -----
Clone     Method
CompareTo Method
Contains  Method
CopyTo   Method
Endswith Method
Equals   Method
GetEnumerator Method
GetHashCode Method
GetType   Method
Definition
-----
System.Object Clone(), System.Object ...
int CompareTo(System.Object value), i...
bool Contains(string value)
void CopyTo(int sourceIndex, char[] d...
bool Endswith(string value), bool End...
bool Equals(System.Object obj), bool ...
System.CharEnumerator GetEnumerator()
int GetHashCode()
type GetType()


```

Figure 9-14

When you’re ready, you can store your results to a file, by using **Out-File**.

```
Get-Process | ConvertTo-Csv | Out-File -FilePath c:\Process.csv
```

It's important to remember that **ConvertTo-** cmdlets leave the objects in the pipeline and the **Export-** cmdlets do not. Export last.

## Converting objects to HTML

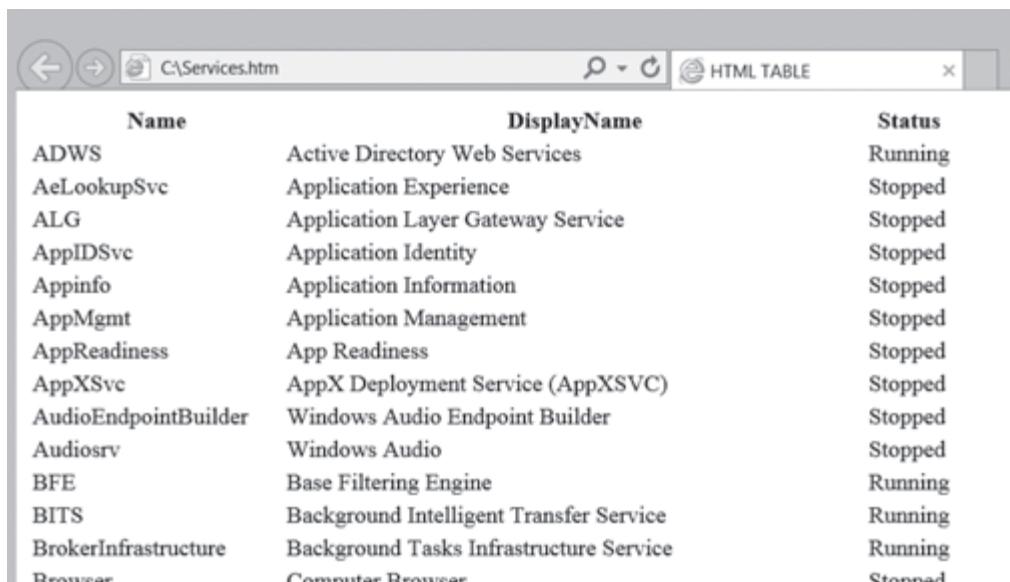
PowerShell includes the **ConvertTo-HTML** cmdlet, which converts output to an HTML table. This is by far one of our favorite cmdlets—as we can easily create web reports. At its simplest, you can create a report by doing the following:

```
Get-Service | ConvertTo-HTML | Out-file c:\Services.htm
```

Now when you open Services.htm in a browser, you'll see a pretty complete table of running services and their properties. By default, the cmdlet lists all properties. However, you can specify the properties by name and in whatever order you prefer by using **Select-Object** or by using the **-Property** parameter on the **ConvertTo-HTML** cmdlet

```
Get-Service | ConvertTo-HTML -Property Name, DisplayName, Status | Out-File services.html
```

```
Get-Service | Select-Object -Property Name, DisplayName, Status | ConvertTo-Html | Out-File c:\Services.htm
```



The screenshot shows a Microsoft Edge browser window displaying a table of service information. The title bar says 'C:\Services.htm'. The table has three columns: 'Name', 'DisplayName', and 'Status'. The data is as follows:

Name	DisplayName	Status
ADWS	Active Directory Web Services	Running
AeLookupSvc	Application Experience	Stopped
ALG	Application Layer Gateway Service	Stopped
AppIDSvc	Application Identity	Stopped
Appinfo	Application Information	Stopped
AppMgmt	Application Management	Stopped
AppReadiness	App Readiness	Stopped
AppXSvc	AppX Deployment Service (AppXSVC)	Stopped
AudioEndpointBuilder	Windows Audio Endpoint Builder	Stopped
Audiosrv	Windows Audio	Stopped
BFE	Base Filtering Engine	Running
BITS	Background Intelligent Transfer Service	Running
BrokerInfrastructure	Background Tasks Infrastructure Service	Running
Computer Browser	Computer Browser	Stopped

Figure 9-15

Now when you open Services.htm, it's a little easier to work with. As a side note—**Out-File** supports a Universal Naming Convention (UNC) as the path, so you could send the .htm file directly to a web server. This is a great solution when you want other admins to check the report. As an example, we want to obtain a list of all the latest error messages in the system log and put them into a web page. The command might look something like the following:

```
Get-EventLog -LogName System -Newest 2 -EntryType Error | Select TimeGenerated, Index, Message | ConvertTo-HTML | Out-File \\SAPIENDC\C$\inetpub\wwwroot\error.htm
```

```
Start iexplore http://SapienDC/Error.htm
```

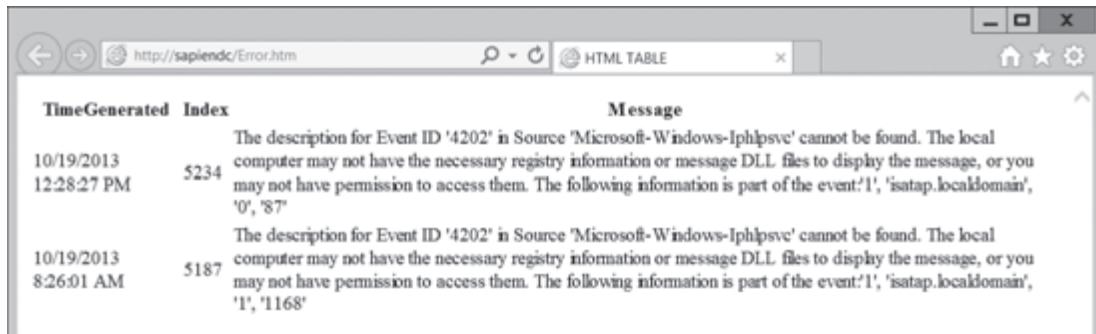


Figure 9-16

We used **Get-EventLog** to obtain a list of errors, converted it to HTML, and then created the file on the default website of a Microsoft IIS web server. The last line simply starts iexplore.exe with the web address.

## Dressing it up

If you want to dress up the page a bit, **ConvertTo-HTML** has some additional parameters.

## ConvertTo-HTML parameters

### ConvertTo-HTML

Head

### Optional parameters

Title

Inserts text into the <head> tag of the HTML page. This is useful if you want to include metadata or style sheet references.

Body

Inserts text into the <title> tag of the HTML page. This lets you have a more meaningful title to the page other than the default HTML TABLE.

CssUri

Inserts text within the <body></body> tag. This lets you specify body formatting like fonts and colors, as well as any text you want to appear before the table.

PreContent

Lets you specify the location of a Cascading Style Sheet (CSS), which will be used to apply formatting to the HTML output.

PostContent

Content to be prepended to the final output.

Content to be appended to the final output.

---

### Note:

As always, you can also find the full details yourself by reading the Help file.

We're going to show you a lot of cmdlets like **ConvertTo-HTML**, where we only introduce you to some of the things they can do. You should always read the Help on these cmdlets to see what else they can do. You don't need to memorize the syntax, but knowing what capabilities are in the shell will help you solve real-world problems when the time comes—even if you have to do a bit of research to re-discover something you read about in the past. Let's dress up our previous example a little.

```
Get-EventLog -LogName System -Newest 2 -EntryType Error | Select TimeGenerated, Index, Message | ConvertTo-Html -Title "Error Report" -PreContent "<p><b>Daily Error Report</b></p>" -PostContent "<p><b>_End_</b></p>" | Out-File c:\error.htm
```

TimeGenerated	Index	Message
10/19/2013 12:28:27 PM	5234	The description for Event ID '4202' in Source 'Microsoft-Windows-Iphlpsvc' cannot be found. The local computer may not have the necessary registry information or message DLL files to display the message, or you may not have permission to access them. The following information is part of the event:'1', 'isatap.localdomain', '0', '87'
10/19/2013 8:26:01 AM	5187	The description for Event ID '4202' in Source 'Microsoft-Windows-Iphlpsvc' cannot be found. The local computer may not have the necessary registry information or message DLL files to display the message, or you may not have permission to access them. The following information is part of the event:'1', 'isatap.localdomain', '1', '1168'

**\_End\_**

Figure 9-17

There is plenty you can do dress up your reports even further if you research a little about CSS.

## Formatting object for reports

We don't always want to store data as web page report or convert it to CSV or XML. Sometimes, a simple, nice-looking text report is all we need. PowerShell doesn't always display or save the information the way we want, so we control the formatting with the **Format-** cmdlets. To see a list of the **Format-** cmdlets, run **Get-Help \*Format-\***.

### Formatting rules overview: When does PowerShell use a list or a table?

PowerShell will automatically format with one of the **Format** cmdlets, depending on a couple of things. First, if PowerShell has a custom output view for the first object in the pipeline, it'll use that view—and the view itself defines whether it's a table, list, custom, or wide view. PowerShell comes with many custom views for many different types of objects.

If there is no custom view, however, PowerShell will use **Format-List** if the first object in the pipeline has five or more properties; otherwise, it'll send the output to **Format-Table**. If **Format-Table** is selected, then it'll use the properties of the first object in the pipeline to form the table columns. So, if the first object has three properties and every other object in the pipeline has ten properties, you'll have a three-column table.

Note that PowerShell's formatting files—or a custom formatting file you create—can contain multiple possible views for a given type of data. Normally, when you explicitly use **Format-Table**, **Format-List**, **Format-Wide**, or **Format-Custom**, PowerShell will select the first registered view that matches that layout option. That is, if you send data to **Format-Table**, PowerShell will find the first table-style view and use it.

All of the formatting cmdlets, however, support a **-View** parameter, which lets you specify an alternate view. For example, **Get-Process** normally returns a view like the one shown in Figure 9-18.

**Get-Process**

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "PS C:\> Get-Process" is run, and the results are displayed in a table view. The columns are Handles, NPM(K), PM(K), WS(K), VM(M), CPU(s), Id, and ProcessName. The data includes various system processes like calc, conhost, csrss, dfssrs, dfssvc, dllhost, dns, dwm, and explorer.

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
113	20	8212	13436	102	0.16	1472	calc
58	7	1896	7808	68	4.02	2372	conhost
44	5	716	3180	44	0.00	2660	conhost
192	12	1664	3776	44	0.78	348	csrss
143	17	1488	21500	61	2.30	412	csrss
340	31	14288	19104	716	0.84	1644	dfssrs
127	12	1772	5228	29	0.00	1928	dfssvc
194	13	3240	10312	48	0.25	2272	dllhost
5261	3696	48216	47600	94	0.39	1696	dns
214	21	31000	52580	141	1.25	788	dwm
952	52	19012	49916	308	2.58	2268	explorer
0	0	0	24	0		0	Idle

Figure 9-18

That's a table view. Yes, it has more than five properties, but it's a view specifically registered for **Process** objects and PowerShell will always use a registered, type-specific view if one is available. However, you could specify an alternate view.

```
Get-Process | Format-Table -View Priority
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "PS C:\> Get-Process | Format-Table -view Priority" is run, and the results are displayed in two separate tables based on PriorityClass. The first table, labeled "PriorityClass: Normal", lists processes like calc, conhost, csrss, dfssrs, dfssvc, dllhost, and dns. The second table, labeled "PriorityClass: High", lists the dwm process. Both tables include columns for ProcessName, Id, HandleCount, and WorkingSet.

ProcessName	Id	HandleCount	WorkingSet
calc	1472	113	13758464
conhost	2372	58	7995392
conhost	2660	44	3256320
csrss	348	189	3862528
csrss	412	143	22016000
dfssrs	1644	333	19542016
dfssvc	1928	127	5353472
dllhost	2272	194	10559488
dns	1696	5259	48734208

ProcessName	Id	HandleCount	WorkingSet
dwm	788	214	53841920

Figure 9-19

Notice that this particular view is grouping **Process** objects on their **PriorityClass** property, and it has defined a different list of columns. We had to explicitly use **Format-Table** (or its alias, **Ft**) because

we needed to specify the view's name, **Priority**. If we'd used another formatting cmdlet, this wouldn't have worked, because the **Priority** view is defined as a table—it can't be selected by anything but **Format-Table**.

Unfortunately, there's no quick or easy way to determine what special formats are available in PowerShell's built-in formatting files, other than by opening them up in Notepad and browsing them. You'll find them in PowerShell's installation folder, each with a .format.ps1xml filename extension.

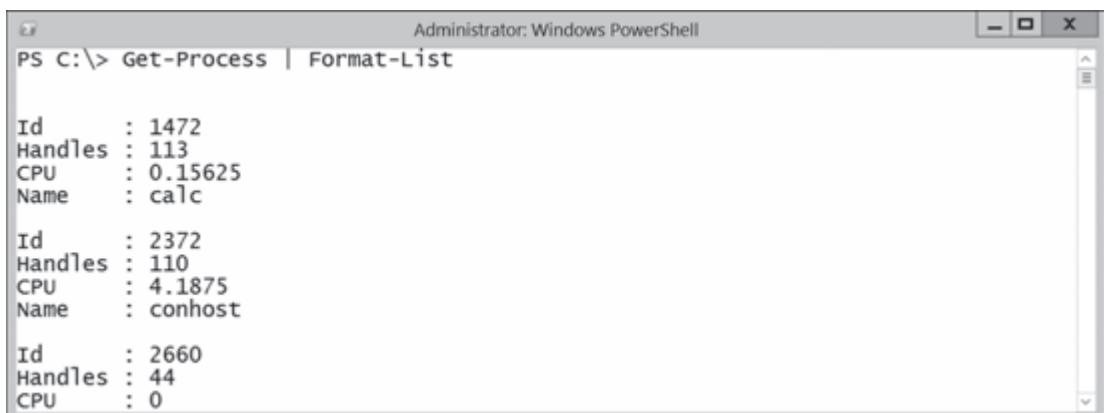
## Formatting lists and tables

Just about every PowerShell cmdlet is designed to produce textual output. The cmdlet developer creates a default output format based on the information to be delivered. For example, the output of **Get-Process** is a horizontal table. However, if you need a different output format, PowerShell has a few choices that are discussed below.

### Format-List

This cmdlet produces a columnar list. Here's a sample using **Get-Process**.

```
Get-Process | Format-List
```



```
Administrator: Windows PowerShell
PS C:\> Get-Process | Format-List

Id      : 1472
Handles : 113
CPU     : 0.15625
Name    : calc

Id      : 2372
Handles : 110
CPU     : 4.1875
Name    : conhost

Id      : 2660
Handles : 44
CPU     : 0
```

Figure 9-20

Even though we've truncated the output, you get the idea. Instead of the regular horizontal table, PowerShell lists each process and its properties in a column. As we've pointed out previously, **Format-List** doesn't use all the properties of an object—PowerShell instead follows its internal formatting rules, which may include using a view—essentially a set of pre-selected properties. Microsoft provides views for most of the common types of information you'll work with, and in many cases provides both table- and list-style views so that PowerShell doesn't just pick random properties for its output.

If you prefer more control over what information is displayed, you can use the **-Property** parameter to specify the properties.

```
Get-Process -Name MSPaint | Format-List -Property Name, WorkingSet, ID, Path
```

```
Administrator: Windows PowerShell
PS C:\> Get-Process -Name MSPaint | Format-List -Property Name, WorkingSet, ID, Path

Name      : mspaint
WorkingSet : 20074496
Id        : 1480
Path      : C:\Windows\system32\mspaint.exe

PS C:\>
```

Figure 9-21

In Figure 9-21, we've called **Get-Process**, seeking specific information on the MSPaint process.

### How did we know?

You might wonder how we knew what properties PowerShell can display when we use the **-Property** cmdlet. To review, it is important to learn about the **Get-Member** cmdlet. This command lists all the available properties for the process object.

```
Get-Process | Get-Member
```

Different cmdlets and objects have different properties, especially in WMI.

Note that if you don't specify any properties, **Format-List** either uses a pre-defined view, or if none exists, **Format-List** looks in PowerShell's Types.ps1xml type definition file to see if any properties are marked as defaults. If some properties are marked as defaults, then PowerShell will use those properties to construct the list. Otherwise, PowerShell uses all of the object's properties to construct the list. To force the cmdlet to list all of an object's properties, use **\***.

```
Get-Process | Format-List *
```

### Format-Table

Just as there are some cmdlets that use a list format as the default, there are some that use a table format. Of course, sometimes you may prefer a table. Some cmdlets produce results in a list by default, such as the **Get-WmiObject** cmdlet. Why does this cmdlet produce a list by default? There are two possibilities: Microsoft provided a pre-defined view, which happens to be a list, or Microsoft provided no pre-defined view. In the latter case, PowerShell selected the list type because the object has more than five properties—if it had fewer, PowerShell would select a table under the same circumstances.

## Exporting and formatting your data

```
Get-WmiObject -Class Win32_logicaldisk
```

A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-WmiObject -Class Win32\_logicaldisk" has been run. The output shows two logical disks, A: and C:, with their properties listed in a plain text format.

Property	Value
DeviceID	: A:
DriveType	: 2
ProviderName	:
FreeSpace	:
Size	:
VolumeName	:
DeviceID	: C:
DriveType	: 3
ProviderName	:
FreeSpace	: 51118301184
Size	: 64055406592
VolumeName	:

Figure 9-22

This is not too hard to read. However, here's the same cmdlet using the **Format-Table** cmdlet.

```
Get-WmiObject -Class Win32_logicaldisk | Format-Table DeviceID,
@{ n = 'FreeSpace'; e = {$_._Freespace / 1Gb -as [int] } },
@{ n = 'Size'; e = {$_._Size / 1Gb -as [int] } }
```

A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-WmiObject -Class Win32\_logicaldisk | Format-Table DeviceID, @n='FreeSpace';e={\$\_.\_Freespace / 1Gb -as [int]}}, @n='Size';e={\$\_.\_Size / 1Gb -as [int]}"} has been run. The output is a table with columns for DeviceID, FreeSpace, and Size.

DeviceID	FreeSpace	Size
A:	0	0
C:	48	60
D:	0	0
Z:	223	699

Figure 9-23

Notice that the property headings are in the same order that we specified in the expression. They also use the same case. You can use \* for the property list. However, doing so will often create unreadable output, because PowerShell tries to squeeze all the properties into a single screen-width table, and most objects simply have too many properties for that to work out well.

You will also notice that we created our own custom properties to make the numbers (in bytes) more readable. Refer to Chapter 6 for a review of those.

**Format-Table** lets you tweak the output by using **-AutoSize**, which automatically adjusts the table.

```
Get-WmiObject -Class Win32_logicaldisk | Format-Table DeviceID,
@{ n = 'FreeSpace'; e = ${_.Freespace / 1Gb -as [int]} },
@{ n= 'Size'; e = ${_.Size / 1Gb -as [int]} } -AutoSize
```

```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject -Class Win32_logicaldisk | Format-Table DeviceID,
>> @{n='FreeSpace';e=${_.Freespace / 1Gb -as [int]}},
>> @{n='Size';e=${_.Size / 1Gb -as [int]}} -AutoSize
>>

DeviceID FreeSpace Size
----- ----- --
A: 0 0
C: 48 60
D: 0 0
Z: 223 699

PS C:\> -
```

Figure 9-24

This is the same command as before, except it includes **-AutoSize**. Notice how much neater the output is. Using **-AutoSize** eliminates the need to calculate how long lines will be, add padding manually, or use any scripting voodoo.

## Format-Wide

Some cmdlets, like **Get-Service**, produce a long list of information that scrolls off the console screen. Wouldn't it be nice to view this information in multiple columns across the console screen? We can accomplish this with the **Format-Wide** cmdlet.

```
Get-Service | Format-Wide -Column 3
```

```
Administrator: Windows PowerShell
PS C:\> Get-Service | Format-Wide -Column 3
```

ADWS	AeLookupSvc	ALG
AppHostSvc	AppIDSvc	Appinfo
AppMgmt	AppReadiness	AppXSVC
AudioEndpointBuilder	Audiosrv	BFE
BITS	BrokerInfrastructure	Browser
CertPropSvc	COMSysApp	CryptSvc
DcomLaunch	defragsvc	DeviceAssociationService
DeviceInstall	Dfs	DFSR
Dhcp	DNS	Dnscache
dot3svc	DPS	DsmSvc
DsRoleSVC	Eaphost	EFS
EventLog	EventSystem	fdPHost
FDResPub	FontCache	FontCache3.0.0.0

Figure 9-25

If you prefer more than the default of two columns, use the **-Column** parameter to specify the number of columns.

Unlike **Format-Table** and **Format-List** that allow multiple properties, **Format-Wide** permits only a single property.

## Grouping information

All the **Format-** cmdlets include the **-GroupBy** parameter, which allows you to group output based on a specified property. For example, here is a **Get-Service** expression that groups services by their status, such as **Running** or **Stopped**.

```
Get-Service | Format-Table -GroupBy Status
```

```
Administrator: Windows PowerShell
PS C:\> Get-Service | Format-Table -GroupBy Status
```

Status: Running		
Status	Name	DisplayName
Running	ADWS	Active Directory Web Services

Status: Stopped		
Status	Name	DisplayName
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service

Figure 9-26

As you can see, grouping helps a little bit. However, this is probably not what you expected, since the cmdlet basically just generates a new group header each time it encounters a new value for the specified property. Since the services weren't first sorted on that property, they aren't grouped like you might want them to be. So, the trick, prior to grouping, is to sort them, first.

```
Get-Service | Sort-Object -Property Status | Format-Table -GroupBy Status
```

### By the way...

Using the **-GroupBy** parameter is the same as piping objects to the **Group-Object** cmdlet, and then piping the objects to a **Format-** cmdlet. In both cases, you'll want to sort the objects first by using **Sort-Object**.

### The grid view

No discussion about formatting your data would be complete without one of our favorite cmdlets: **Out-GridView**.

```
Get-Service | Out-GridView
```

Status	Name	DisplayName
Running	ADWS	Active Directory Web Services
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Running	AppHostSvc	Application Host Helper Service
Stopped	ApplDSvc	Application Identity
Stopped	Appinfo	Application Information
Stopped	AppMgmt	Application Management
Stopped	AppReadiness	App Readiness
Stopped	AppXSvc	AppX Deployment Service (AppXSVC)

Figure 9-27

You can sort the resulting grid on any column by clicking the column header. You can also search for text strings within any column by simply typing your text at the top of the window and pressing **Enter**; non-matching rows are removed from the display.

Get-Service | Out-GridView

Stopped

Add criteria ▾

Status	Name	DisplayName
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Stopped	ApplDSvc	Application Identity
Stopped	Appinfo	Application Information
Stopped	AppMgmt	Application Management
Stopped	AppReadiness	App Readiness
Stopped	AppXSvc	AppX Deployment Service (AppXSVC)
Stopped	AudioEndpointBuilder	Windows Audio Endpoint Builder
Stopped	Audiosrv	Windows Audio

Figure 9-28

You can create a complex filter by clicking **Add criteria**. If you don't see that button, click the down pointing arrow next to the text input. Select the properties you want to filter on, click **Add**, and then modify the filter. You'll notice that the criteria operator is underlined. Click on it and select the criteria you would like to use. Add, delete, and edit criteria as needed. Click the magnifying glass icon to apply the filter.

Get-Service | Out-GridView

Filter

and Status contains stopped x

and Name contains b x

Add criteria ▾ x Clear All

Status	Name	DisplayName
Stopped	AudioEndpointBuilder	Windows Audio Endpoint Builder
Stopped	Browser	Computer Browser
Stopped	FDResPub	Function Discovery Resource Publicati...
Stopped	vmicheartbeat	Hyper-V Heartbeat Service
Stopped	WPDBusEnum	Portable Device Enumerator Service

Figure 9-29

You can restore the original, full results by clearing your search text and pressing **Enter** again, or by removing the search criteria.

**Out-GridView** also has an interesting parameter, **-PassThru**. You can sort and select the data you want from the grid view, then when you click **OK**, only that data is passed down the pipeline.

```
Get-Service | Out-GridView -PassThru | format-table
```

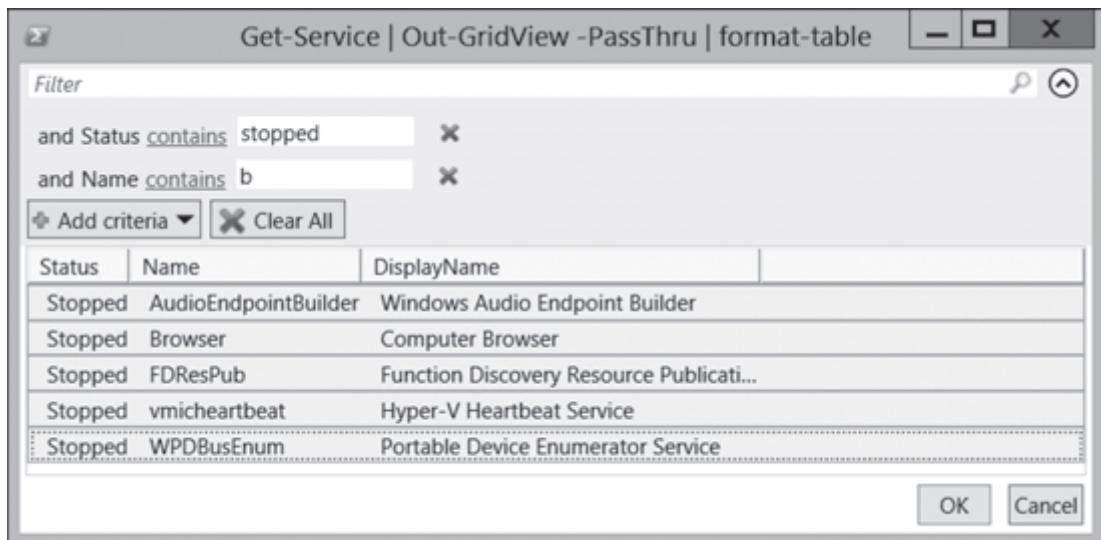


Figure 9-30

In the Figure 9-30, we selected only the services that contain the letter “b” and that have a status of stopped. When we click **OK**, those services are sent down the pipeline.

Status	Name	DisplayName
Stopped	AudioEndpointBu...	Windows Audio Endpoint Builder
Stopped	Browser	Computer Browser
Stopped	FDResPub	Function Discovery Resource Publica...
Stopped	vmicheartbeat	Hyper-V Heartbeat Service
Stopped	WPDBusEnum	Portable Device Enumerator Service

Figure 9-31

We just sent them to **Format-Table**—but you could just as easily pipe the results to **Start-Service**.

**Out-GridView**, which has an alias of **ogv**, is especially useful for reviewing larger result sets in a scrollable, resizable table. Keep in mind that the view is static; it does not continually update. To refresh the grid, you need to close the grid window, and then re-execute your original command.

## Exercise 9 – Exporting and formatting your data

Time: 30 minutes

This exercise gives you a chance to try out the **Export-** and **ConvertTo-** cmdlets, along with previous skills you've learned.

### Task 1

Create an HTML report that contains the 10 highest-running CPU process, listed highest to smallest. Select only the process name, the CPU, and the ID.

### Task 2

Create an XML list of all running process and save this as c:\Good.xml. Start Notepad, Calc, and a few other applications. Can you compare the known good-process list to the current running processes?

### Task 3

Run the following:

```
Get-HotFix | Export-Csv MYHotfix.csv | Out-File MyFile.csv
```

What happens and why?

### Task 4

Using only a format command, obtain a list of process and format the results to a table with the properties Name, CPU, ID, VM, PM, WS. Create custom properties and convert VM, PM, and WS to megabytes.

### Task 5

Create a table that groups stopped services, then running services.

### Task 6

When using **Out-File**, you want to make sure that if there is an existing file, it is not overwritten. How would you accomplish this?

## Chapter 10

# Automation: Your first script

### Getting ready for your first script

With many shells—particularly some \*nix shells—using the shell interactively is a very different experience than scripting with the shell. Typically, shells offer a complete scripting language that is only available when you’re running a script. Not so with PowerShell. The shell behaves the same and offers the same features and functionality, whether you’re writing a script or using the shell interactively. In fact, a PowerShell script is a true script—simply a text file listing the things you’d type interactively. The only reason to write a script is because you’re tired of typing those things interactively and want the ability to run them again and again, with minimal effort.

### Script files

PowerShell recognizes the .PS1 filename extension as a PowerShell script. Script files are simple text files and you can edit them by using Windows Notepad or any other text editor. In fact, by default, Windows associates the .PS1 filename extension with Notepad, not PowerShell, so double-clicking a script file opens it in Notepad rather than executing it in PowerShell. Of course, we’re a bit biased against Notepad as a script editor—Notepad was certainly never designed for that task and better options exist. There is a built-in learning tool—the Integrated Script Environment (ISE)—to get you started. When you’re ready for a more professional scripting environment (similar to Visual Studio), there are professional options. We’re obviously keen on SAPIEN PrimalScript ([www.Sapien.com](http://www.Sapien.com)) and SAPIEN PowerShell Studio because they offer a full visual development environment with PowerShell specific support, such as the ability to package a PowerShell script in a standalone executable that runs under alternate credentials and to create graphical Windows applications.

Note that PowerShell version 4 continues to use the .PS1 filename extension. Think of this filename extension as “version 1 of the script file format,” rather than having a direct relation to version 4 of the shell.

## Scripting security features

Before you create and run your first script, it pays to take a little time and understand a little about script security. PowerShell has some interesting challenges to meet in terms of security. The last time Microsoft produced a scripting shell, it produced the Windows Script Host (WSH, which is commonly used to run VBScript scripts). When WSH was produced, security wasn’t much of a concern, which meant that VBScript probably became one of the biggest attack points within Windows. As a result, it was used to write and execute a number of viruses and other malicious attacks. Microsoft certainly didn’t want to repeat that with PowerShell, and so a number of security features have been built-in.

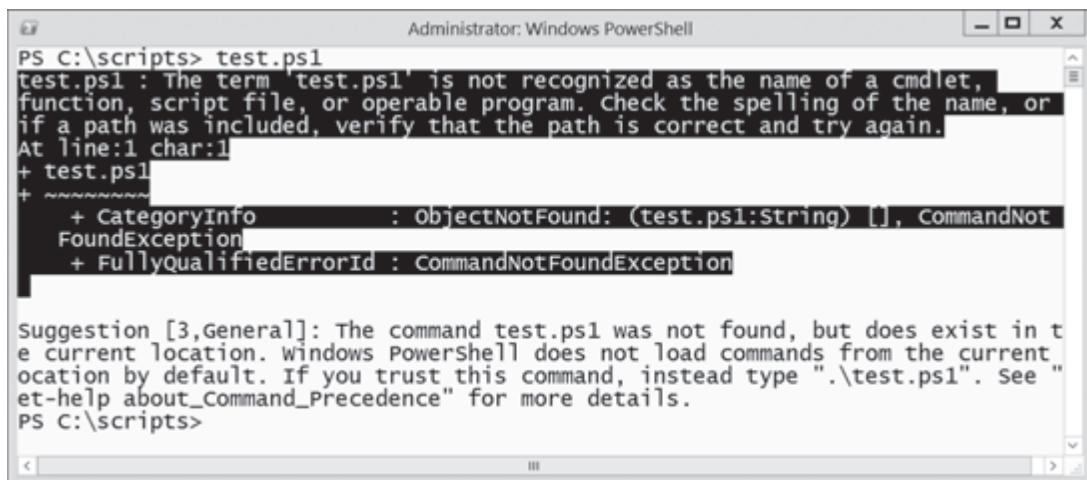
### Why won’t my scripts run?

The first thing you may notice is that files having a .PS1 filename extension don’t do anything automatically when double-clicked. The PowerShell window might not even open, since by default the .PS1 filename extension is associated with Notepad! Even if you manually open PowerShell and type a script name, it doesn’t run. What good is PowerShell if it can’t run scripts?

### When scripts don’t run

There are two reasons why a PowerShell script might not run. The first is the script’s location; the second is you don’t have permissions to run the script by default. PowerShell won’t run scripts that are located in the current directory—it’s as if it simply can’t see them. For example, we created a script named Test.ps1 in a folder named C:\scripts. With PowerShell set to that folder, we type the script name—it isn’t found.

```
PS C:\scripts> test.ps1
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\scripts> test.ps1". The output shows an error message: "test.ps1 : The term 'test.ps1' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again." Below this, detailed error information is shown, including stack traces for CategoryInfo, ObjectNotFoundException, and FullyQualifiedErrorId. At the bottom, a suggestion is provided: "Suggestion [3,General]: The command test.ps1 was not found, but does exist in t e current location. Windows PowerShell does not load commands from the current ocation by default. If you trust this command, instead type ".\test.ps1". See "et-help about\_Command\_Precedence" for more details."

Figure 10-1

We receive an error message: “The term ‘test.PS1’ is not recognized as a cmdlet, function, operable program, or script file. Verify that the path is correct and try again.”

You will also notice that a helpful suggestion is displayed to provide additional guidance. If you read it, it tells you how to fix it—but wait—let’s look at why this is happening.

This is a security precaution to help prevent a malicious script from intercepting cmdlet and command names, and then running. In other words, if we named our script Dir.PS1, it still wouldn’t run—the **Dir** command would run instead. So, the only way to run a script is to explicitly refer to that folder.

```
PS C:\scripts> c:\Scripts\Test.ps1
```

Or, if you are in the current folder with the script.

```
PS C:\Scripts>.\Test.ps1
```

Well, if we try this, we are still in for a rude message.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is ".\test.ps1". The output is as follows:

```
PS C:\scripts> .\test.ps1
.\test.ps1 : File C:\scripts\test.ps1 cannot be loaded because running scripts
is disabled on this system. For more information, see about_Execution_Policies
at http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\test.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: () [], PSInvalidOperationException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\scripts> -
```

Figure 10-2

This is a second reason scripts may not run and it is another security precaution. The following error is displayed: “C:\Scripts\Test.Ps1 cannot be loaded because running scripts is disabled on this system.”

By default, scripts are not permitted to run. For this to be corrected, we need to choose a script execution policy.

## What's an execution policy?

The second reason your script may not run is the execution policy. For security purposes, PowerShell defaults to a very restrictive execution policy that says the shell can only be used interactively, which occurs when you type in commands directly and have them execute immediately. This helps ensure that PowerShell can't be used to run script-based viruses, by default. In order to run scripts, you need to change PowerShell to a different execution policy. However, first we need to talk a bit about how PowerShell identifies and decides whether to trust scripts.

## Execution policies

Within PowerShell, you can run `Help for About_signing` to learn more about PowerShell's six execution policies.

- **AllSigned.** In this mode, PowerShell executes scripts that are trusted, which means they must be properly signed. Malicious scripts can execute, but you can use their signature to track down the author.
- **Restricted.** This is the default policy. The restricted mode means that no scripts are executed, regardless of whether they have been signed.
- **RemoteSigned.** In this mode, PowerShell will run local scripts without them being signed. Remote scripts that are downloaded through Microsoft Outlook, Internet Explorer, and so forth must be trusted in order to run.
- **Unrestricted.** PowerShell runs all scripts, regardless of whether they have been signed. Downloaded scripts will prompt before executing to make sure you really want to run them.
- **Bypass.** This execution policy is designed for configurations in which PowerShell is the foundation for a program that has its own security model. This basically allows anything to run.
- **Undefined.** If the execution policy in all scopes is **Undefined**, the effective execution policy is **Restricted**, which is the default execution policy.

We highly recommend that you sign your scripts, because it creates a more secure and trustworthy environment. If you plan to sign your scripts as recommended, then the **AllSigned** execution policy is appropriate. Otherwise, use **RemoteSigned**. The **Unrestricted** policy is overly generous and leaves your computer open to a range of attacks, which is why it shouldn't be used.

You should bear in mind the differences between the **AllSigned** and **RemoteSigned** execution policies. When you download a file with Microsoft Outlook or Microsoft Internet Explorer, Windows marks the file as having come from a potentially untrustworthy source: the Internet. Files marked in this fashion won't run under the **RemoteSigned** execution policy unless they've been signed. While we still encourage the use of the **AllSigned** execution policy, **RemoteSigned** at least lets you run unsigned scripts that you write yourself, while providing a modicum of protection against potentially malicious scripts you acquire from somewhere else.

## Setting an execution policy

To check the current execution policy from within PowerShell, run the following:

```
Get-ExecutionPolicy
```

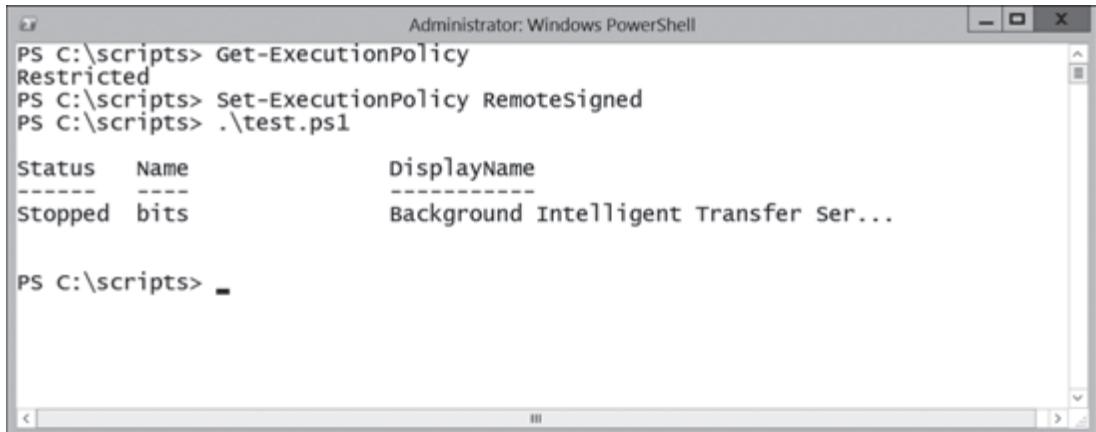
You'll receive information about the current execution policy. If it's **Restricted**, then you know why your scripts won't run.

You can change the execution policy within PowerShell. Keep in mind that this is changing a value in the system registry, to which only admins may have access. Therefore, if you're not an admin on your computer, then you may not be able to modify the execution policy. To change the execution policy, run the following:

```
Set-ExecutionPolicy RemoteSigned
```

This will set the execution policy to **RemoteSigned**. This change will take effect immediately, without restarting PowerShell.

You can also use the group policy **Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell** to set the execution policy for a group of computers.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-ExecutionPolicy" is run, showing the output "Restricted". Then, the command "Set-ExecutionPolicy RemoteSigned" is run. Finally, ".\test.ps1" is executed. A table follows, showing the status and name of the bits service, with its display name being "Background Intelligent Transfer Ser...".

Status	Name	DisplayName
Stopped	bits	Background Intelligent Transfer Ser...

Figure 10-3

In Figure 10-3, we changed the execution policy to **RemoteSigned** and tried our simple script again.

## Digitally signing scripts

Digital signatures are an important part of how PowerShell's security works. A digital signature is created by using a code-signing certificate, sometimes referred to as a Class 3 digital certificate or an Authenticode certificate, which is a Microsoft brand name. These certificates are sold by commercial certification authorities (CA). The certificates can also be issued by a company's own private CA, such as Microsoft's Active Directory Certificate Services.

The trust process starts with the CA. All Windows computers have a list of trusted root CAs that is configured in the **Internet Options** control panel applet, as shown in Figure 10-4. To access this window, open **Internet Options**, and then click **Publishers...** on the **Content** tab. Then, select the **Trusted**

**Root Certification Authorities** tab. This list, which is pre-populated by Microsoft and can be customized by admins through **Group Policies**, determines the CAs that your computer trusts. By definition, your computer will trust any certificates issued by these CAs or any lower-level CA that a trusted CA has authorized. For example, if you trust CA 1, and they authorize CA 2 to issue certificates, then you'll trust certificates issued by CA 1 and by CA 2—your trust of CA 2 comes because it was authorized by the trusted CA 1.

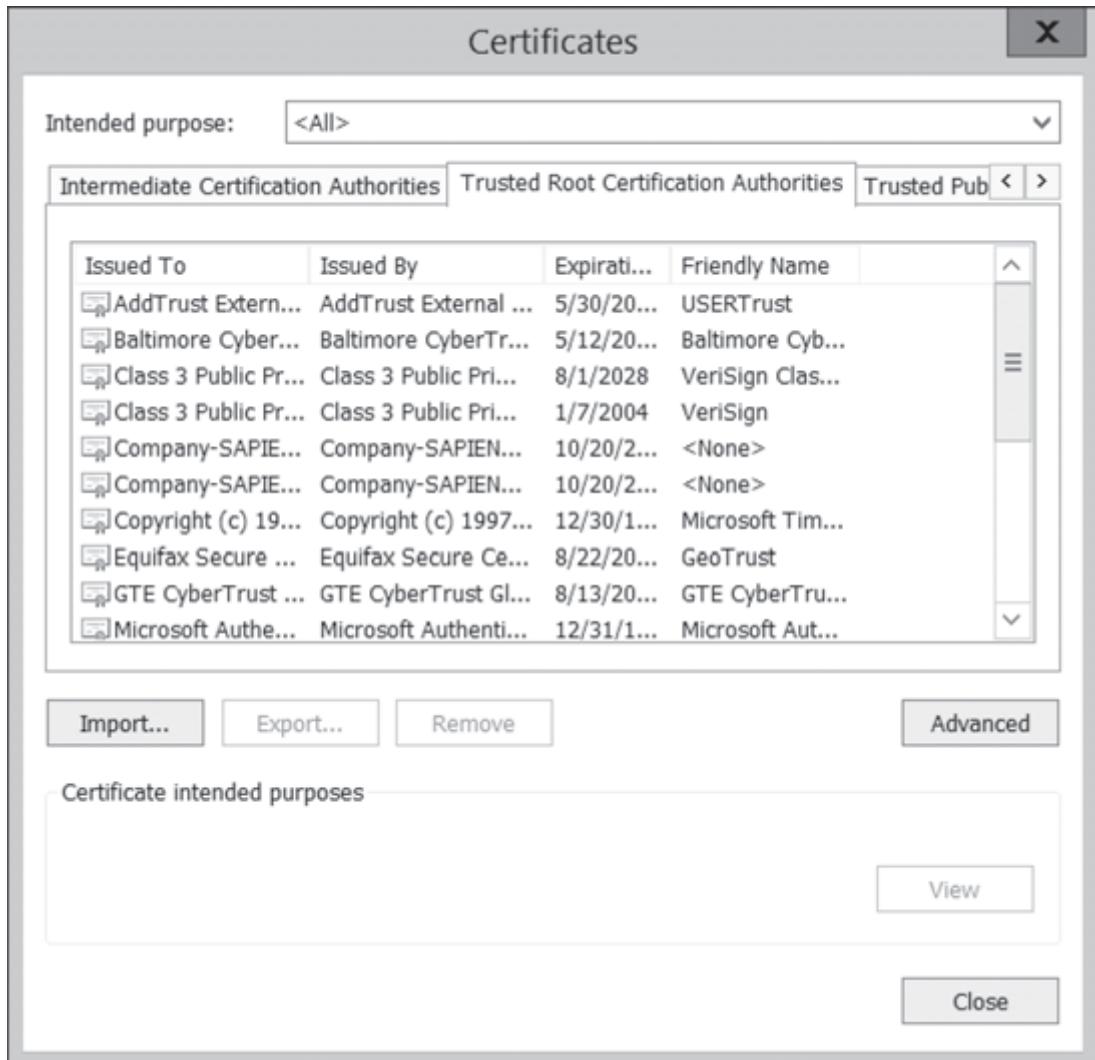


Figure 10-4

When a CA issues a code-signing certificate, it consists of two halves: a private key and a public key. You usually install the entire certificate on your local computer and use it to digitally sign code, including PowerShell scripts. A digital signature is created by calculating a cryptographic hash, which is a kind of complex checksum, on the script's contents. The hash is the result of a complex mathematical algo-

rithm that is designed so that no two different scripts can ever produce the same hash value. In other words, the hash acts as a sort of electronic fingerprint for your script. The hash is then encrypted using your certificate's private key. This encrypted hash, which is referred to as the signature, is appended to the script.

Since the hash portion of the digital signature is unique to your script, it will change if your script changes in the slightest. Even an extra blank line somewhere in your script will invalidate the old hash and digital signature. After making any changes to your script, you need to re-sign it. While you can sign a script manually with PowerShell—see the `About_Signing` Help topic—you can configure tools like SAPIEN PrimalScript and PowerShell Studio to automatically sign scripts each time they're saved, which can save you a lot of hassle.

## Trusted scripts

When PowerShell tries to run a script, it first looks for a signature. If it doesn't find one, then the script is considered untrusted. If PowerShell does find a signature, it looks at the unencrypted part of the signature that contains information about the author of the script. PowerShell uses this information to retrieve the author's public key, which is always available from the CA that issued the code-signing certificate that was used to sign the script. If the CA isn't trusted, then the script isn't trusted. In this case, PowerShell doesn't do anything else with the script or signature.

If the CA is trusted and PowerShell is able to retrieve the public key, then PowerShell tries to decrypt the signature using that public key. If it's unsuccessful, then the signature isn't valid and the script is untrusted. If the signature can be decrypted, then PowerShell knows the script is conditionally trusted, which means it's been digitally signed by a trusted certificate issued by a trusted CA. Finally, PowerShell computes the same hash on the script to see if it matches the previously encrypted hash from the signature. If the two match, PowerShell knows the script hasn't been modified since it was signed, and then the script is fully trusted. If the hashes do not match, then the script has been modified and the script is untrusted because the signature is considered broken.

PowerShell uses the script's status as trusted or untrusted to decide whether it can execute the script in accordance with its current execution policy.

## Digitally signing scripts

Because code-signing certificates can be expensive, you may wish to create a self-signed certificate for your own local testing purposes. This certificate will be trusted only by your personal computer, but it costs you nothing to create. To create a self-signed certificate, you need the program `Makecert.exe`, and it is available in the downloadable Microsoft .NET Framework Software Development Kit (SDK). This file is also downloadable from the Windows Platform SDK.

After downloading and installing this file, you can use the `Makecert.exe` file to create the certificate by running the following from a `Cmd.exe` shell.

```
Makecert -n "CN=PowerShell Local Certificate Root" -a sha1 -eku 1.3.6.1.5.5.7.3.3 -r -sv root.pvk
root.cer -ss Root -sr localMachine
```

**Note:**

If you have problems running Makecert, run Makecert /? to verify the correct syntax. Different versions of Makecert (a version might be included in your Microsoft Office installation, for example) require slightly different command-line arguments.

Next, run the following:

```
Makecert -pe -n "CN=PowerShell User" -ss MY -a sha1 -eku 1.3.6.1.5.5.7.3.3 -iv root.pvk -ic root.cer
```

Again, this is all one long line of typed instructions. These lines create two temporary files, root.pvk and root.cer, that you can save as backups. The actual certificate will be installed into your local certificate store where it can be used to sign scripts. Within PowerShell, run the following:

```
Set-AuthenticodeSignature "filename.ps1" @(Get-ChildItem cert:\CurrentUser\My -CodeSigning)[0]
```

This is also one long line. This line retrieves your code-signing certificate and signs the file named filename.ps1, which should be an existing, unsigned PowerShell script. You can run Help for **Set-Authenticodesignature** for additional help with signing scripts.

We need to emphasize that a certificate made with Makecert is only useful for testing on your local computer. If you want to distribute your scripts internally or externally, you need to acquire a real code-signing certificate.

## Is PowerShell dangerous?

The answer is that PowerShell is no more dangerous than any other application. Certainly, PowerShell has the potential for great destruction, since it can delete files, modify the registry, etc. However, so can any other application. If you run PowerShell as a local admin, then it will have full access to your system—just like any other application. If you follow the principle of least privilege, which means you don't routinely log on to your computer as a local admin and you don't routinely run PowerShell as a local admin, then its potential for damage is minimized—just like any other application. In fact, when set to its **AllSigned** execution policy, PowerShell is arguably safer than many applications, since you can ensure that only scripts signed by an identifiable author will actually be able to run.

Naturally, much of PowerShell's security begins and ends with you. Microsoft has configured it to be very safe out-of-the-box. Therefore, anything you do from there can potentially loosen PowerShell's security. For this reason, before you do anything, you need to understand that your actions could have consequences.

## Safer scripts from the Internet

One potential danger point is downloading PowerShell scripts from the Internet or acquiring them from other untrusted sources. While these scripts are a great way to quickly expand your scripting skills, they present a danger if you don't know exactly what they do. Fortunately, Microsoft has provided the **-WhatIf** parameter, which is a very cool way to find out what scripts do.

All PowerShell cmdlets are built from the same basic class, or template, which allows them to have a **-WhatIf** parameter. Not every cmdlet actually implements this, but then not every cmdlet does something potentially damaging. Let's look at a good example of how you might use the **-WhatIf** parameter.

Say you download a script from the Internet and in it you find the following:

```
Get-Process | Stop-Process
```

This runs the **Get-Process** cmdlet and pipes its output to the **Stop-Process** cmdlet. So, this script will have the effect of stopping every process on your computer. Not good. However, if you weren't sure of this output, you could just add **-WhatIf**.

```
Get-Process | Stop-Process -WhatIf
```

```
Administrator: Windows PowerShell
PS C:\> Get-Process | Stop-Process -WhatIf
what if: Performing the operation "Stop-Process" on target "CertEnrollCtrl (339)".
what if: Performing the operation "Stop-Process" on target "certsvr (3624)".
what if: Performing the operation "Stop-Process" on target "conhost (1212)".
what if: Performing the operation "Stop-Process" on target "conhost (2868)".
what if: Performing the operation "Stop-Process" on target "csrss (344)".
what if: Performing the operation "Stop-Process" on target "csrss (408)".
what if: Performing the operation "Stop-Process" on target "dfsrs (1348)".
what if: Performing the operation "Stop-Process" on target "dfssvc (1652)".
what if: Performing the operation "Stop-Process" on target "dllhost (2212)".
what if: Performing the operation "Stop-Process" on target "dns (1404)".
what if: Performing the operation "Stop-Process" on target "dwm (780)".
what if: Performing the operation "Stop-Process" on target "explorer (2892)".
what if: Performing the operation "Stop-Process" on target "Idle (0)".
what if: Performing the operation "Stop-Process" on target "inetinfo (3828)".
```

Figure 10-5

Other than receiving this output, nothing would happen. The **-WhatIf** parameter tells PowerShell (or more specifically, it tells the **Stop-Process** cmdlet) to display what it would do, without actually doing it. This allows you to see what the downloaded script would have done without taking the risk of running it. That's one way to make those downloaded scripts a bit safer in your environment—or at least see what they'd do. Most cmdlets that change the system in some way support **-WhatIf**, and you can check individual cmdlets' built-in Help to be sure.

Note that **-WhatIf** doesn't take the place of a signature. For example, if your PowerShell execution policy is set to only run trusted (signed) scripts, and you download a script from the Internet that isn't signed, then you'll have to sign the script before you can add **-WhatIf** and run the script to see what it would do.

## Using PrimalScript/PowerShell Studio to sign scripts

Manually signing scripts is a pain—but SAPIEN’s PrimalScript and PowerShell Studio make it easy. First, you need to obtain a code signing certificate. For this book, we just installed one from our local CA, which is Microsoft Active Directory Certificate Services. You can test this yourself or obtain one from a trusted authority. Once you have installed the code signing certificate, the rest is easy to set up.

### **S**igning with PrimalScript

If you have SAPIEN’s PrimalScript, configuring the ability to sign your scripts every time you save is a fairly simple process. From the **File** menu, select **Options**.

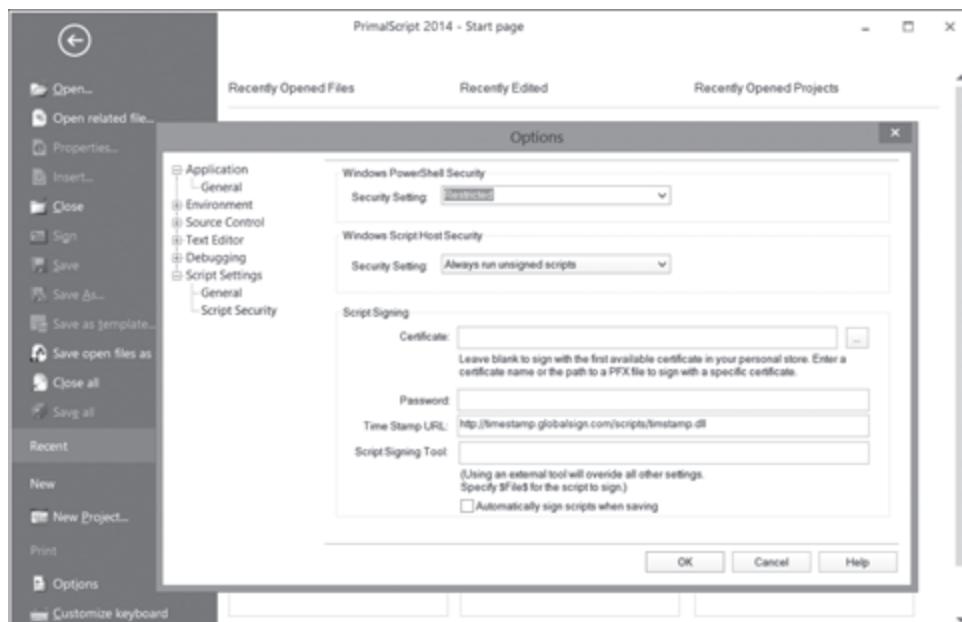


Figure 10-6

In the **Options** window, scroll to **Script Settings**, and then select **Script Security**.

If you have installed the code signing certificate, simply select the **Automatically sign scripts when saving** check box at the bottom of the screen. If you have a .pfx file for your certificate, you can use the ellipsis button to browse for the certificate and supply the .pfx password to add the certificate. When you save your scripts, they will automatically be digitally signed.

### **S**igning scripts with PowerShell Studio

If you have SAPIEN’s PowerShell Studio, configuring the script signing capabilities is very simple. You should have already installed your script or have the .pfx file of your script.

In PowerShell Studio, from the **File** menu, select **Options**.

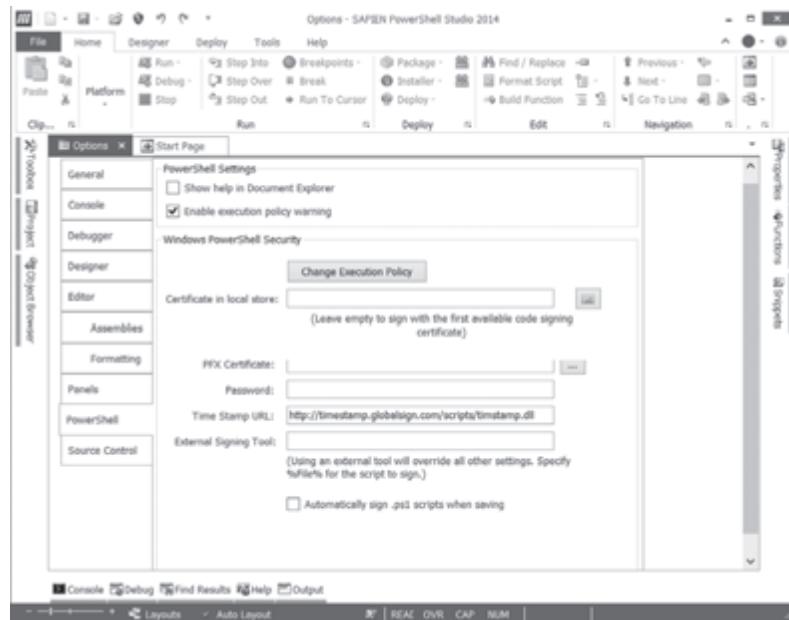


Figure 10-7

The **Options** pane opens to a list of configurable configuration settings for PowerShell Studio. Select the **PowerShell** tab to display configuration options for script signing.

If you have installed the code-signing certificate, simply select the **Automatically sign scripts when saving** and **Automatically sign exported scripts** check boxes. If you have a .pfx file for your certificate, you can use the ellipsis button to browse for the certificate and supply the .pfx password to add the certificate.

Using SAPIEN's PrimalScript or PowerShell Studio to sign scripts makes the process easy. You should consider raising your execution policy to **AllSigned**.

## Turning your one-liners into scripts

Starting in Chapter 13, you will begin to take on the full PowerShell scripting language—but why wait until then to create a useful script? In fact, you already know how to do just that.

Often times you will want to automate (making a simple script) tasks so you don't have to interactively type them in again and again. You could even schedule this script to run using Windows Scheduler or the PowerShell schedule tasks cmdlets. Let's take a moment and make a simple script from one of the useful one-liners demonstrated earlier in the book.

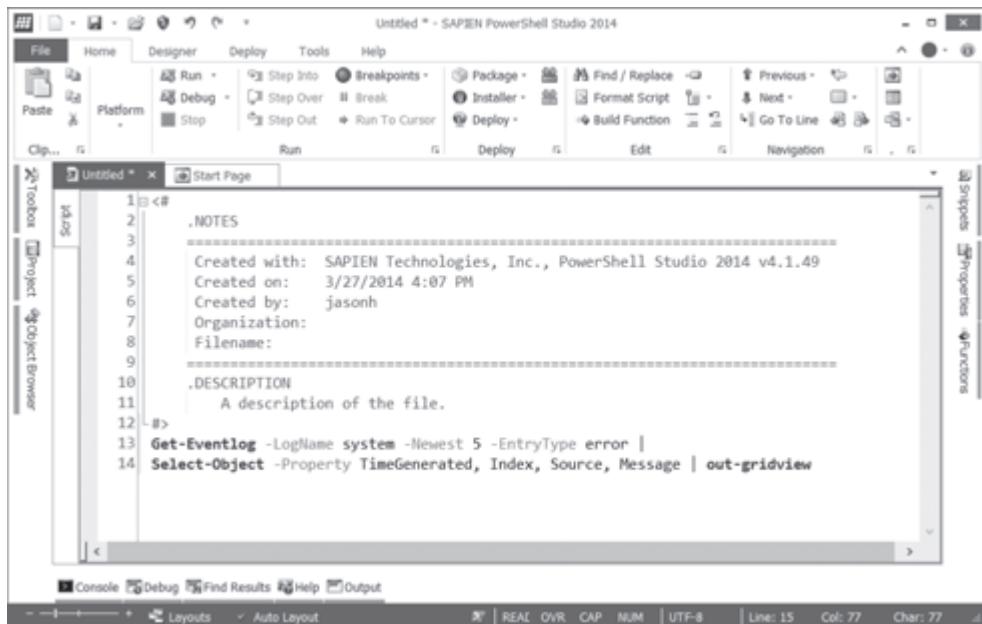
We usually start by trying out our command in the PowerShell console, with the following:

```
Get-Eventlog -LogName system -Newest 5 -EntryType error |
```

```
Select-Object-Property TimeGenerated, Index, Source, Message | out-gridview
```

## Windows PowerShell: TFM

Once we have verified that it works the way we want, then we simply copy it into our favorite script tool. Notice, we also cleaned up the display to make it easier to read. In other words, as you will learn more about later, PowerShell has many line-continuation characters to make your scripts readable and easier to troubleshoot. In the screenshot below, notice that we used the pipeline character as line continuation. The one-liner still runs as if it was one line, but it's easier to read.



The screenshot shows the SAPIEN PowerShell Studio 2014 interface. The main window displays a PowerShell script titled 'Untitled.ps1'. The script content is as follows:

```
1 <#
2 .NOTES
3 -----
4     Created with: SAPIEN Technologies, Inc., PowerShell Studio 2014 v4.1.49
5     Created on: 3/27/2014 4:07 PM
6     Created by: jasonh
7     Organization:
8     Filename:
9 -----
10    .DESCRIPTION
11        A description of the file.
12 >
13 Get-Eventlog -LogName system -Newest 5 -EntryType error |
14 Select-Object -Property TimeGenerated, Index, Source, Message | out-gridview
```

The interface includes a toolbar at the top with various icons for running, debugging, and deploying scripts. The bottom of the window shows tabs for 'Console', 'Debug', 'Find Results', 'Help', and 'Output', along with status information like 'Line: 15' and 'Col: 77'.

Figure 10-8

After saving the script, we like to test the script in the PowerShell console, since that is the location where it will normally be run.

```
PS C:\Scripts> .\test.ps1
```

Out-GridView			
Filter			
Add criteria ▾			
TimeGenerated	Index	Source	Message
10/19/2013 12:28:27 PM	5,234	Microsoft-Windows-Iphlpsvc	The description for Event ID '4202' in Source 'Micro
10/19/2013 8:26:01 AM	5,187	Microsoft-Windows-Iphlpsvc	The description for Event ID '4202' in Source 'Micro
10/18/2013 6:33:19 PM	4,895	Microsoft-Windows-Iphlpsvc	The description for Event ID '4202' in Source 'Micro
10/9/2013 5:42:35 PM	4,611	NETLOGON	The dynamic registration of the DNS record 'Forest'

DNS server IP address: ::

Returned Response Code (RCODE): 0

Figure 10-9

Success! We have just made our first real script. Of course you will be diving much deeper than this throughout the rest of the book, but this is a great starting point.

## The process

Keep the idea of the process we went through in this chapter. Often, you will discover a solution at the PowerShell console, and then automate it by quickly saving it as a script.

- Check the execution policy—**Get-ExecutionPolicy**—and make sure that you have the ability to run scripts.
- Change the execution policy, if needed—**Set-ExecutionPolicy**—to **RemoteSigned** or **AllSigned**.
- Build and test a one-liner in the PowerShell console.
- Copy to a scripting tool of your choice and save the script as a .PS1.
- Test the script in the PowerShell console.

Let's try this in a short exercise, before moving forward!

## Exercise 10 – Automating your first script

**Time:** 20 Minutes

The goal of this exercise is to make sure your execution policy is set correctly and to try out creating your own script. Use the **Set-ExecutionPolicy** cmdlet, and we suggest that you use the **RemoteSigned** policy setting.

Take a few minutes and create a good one-liner in the console. Copy that to your scripting tool and save the script.

Don't forget to test your script from the console to make sure everything is working!

## Chapter 11

# Increasing your management capabilities

### Extending PowerShell's management capabilities

One of PowerShell's greatest strengths is its ability to be extended—by adding additional cmdlets, WMI, and .NET—both by Microsoft product teams, third parties, and even by you!

When PowerShell version 1 shipped, it had 192 cmdlets. Doesn't seem like much, but you could extend your capabilities by accessing the WMI and the .NET framework. The challenge to this approach is that it is difficult for admins unfamiliar with these technologies. So the goal has been to add cmdlets to make our job easier.

---

#### Note:

While cmdlets are the easiest to work with, keep in mind that you are never limited.

Let's start with a concept that you are probably very familiar with—the Microsoft Management Console (MMC).

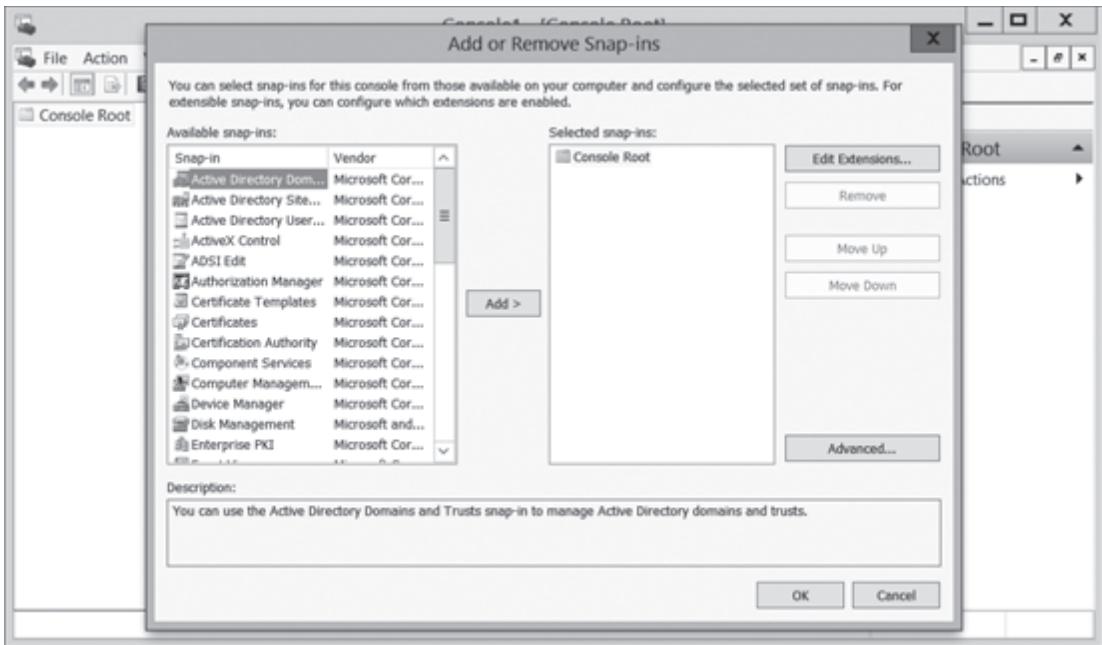


Figure 11-1

When you open the MMC, you need to add snap-ins—additional management tools—to the MMC. Depending on the software you have installed, the snap-in list can be small or large. We have the Remote Server Administration Tools (RSAT) installed, so you will notice management tools such as Active Directory Users and Computers.

This concept is the same in PowerShell, in fact you will hear the term snap-in used. PowerShell allows you to add snap-ins/modules that contain additional cmdlets to the shell. In other words, if you have the RSAT tools or other products installed, you already have the cmdlets you need to manage those products.

### It's a matter of time

Keep something in mind as you work with PowerShell and additional snap-ins and modules. It has taken years for the all the product teams to catch up to the PowerShell management wave—so depending on the version of your product, your operating system, and version of PowerShell, you may not have everything you want. There is just no way around this.

As an example, the Exchange team was one of the first product teams to include cmdlets with their product, Exchange 2007. Now, if you have that product, or a later version, you have everything you need to start managing Exchange with PowerShell (specifically the Exchange Management Shell). Other teams followed suit, such as the Microsoft IIS team, the Active Directory team, and third parties like VMware with PowerCLI. Some teams had a much later start, like System Center, and are providing cmdlets only in recent versions.

Another example is how far we have come recently with the release of Windows Server 2012 R2. Almost all of the roles and features now have modules of cmdlets that you can use to extend PowerShell, over 2000 of them in Windows Server alone! This is an enormous change from Windows Server 2008 R2.

The moral of the story is that more recent software will probably have the cmdlets you're looking for. In writing this book, we are using Windows 8 or 8.1 and Windows Server 2012 (2012 R2). Because of this, we have the luxury of having everything. If you're using older software, keep in mind that some of your options will be limited.

Now, let's start extending PowerShell.

## Snap-ins and modules

Cmdlets are the basic commands available within PowerShell. As such, they encapsulate code that performs useful work. This code is normally written by Microsoft. Cmdlets are normally written in a higher-level .NET Framework language such as Visual Basic, .NET, or C#.

Microsoft initially began implementing collections of cmdlets as snap-ins. Basically, compiled DLLs that are registered on your server or laptop that you can load as needed to gain access to those cmdlets. Examples of products that use snap-ins are Microsoft SQL and Exchange.

When PowerShell version 2 was released, Microsoft added the ability to create modules—much better than snap-ins—mainly to improve the implementation standard and also allow admins like you and me to create our own modules of cmdlets. In fact, later in this book, you will create a module to hold your own cmdlets.

In PowerShell version 2, you had to manually load (import) the modules when you wanted to use them. Because of this, we will show you how to import modules and add snap-ins.

Starting in PowerShell version 3, modules (not snap-ins) will dynamically load when you attempt to use the cmdlet or access Help for a cmdlet. This has vastly improved the discoverability of PowerShell.

```
Get-Help *IPAddress*
```

When you use **Get-Help** to search for something, as in the previous example, PowerShell can now search all installed modules, not just the imported ones, to return results. This is a major feature and reason for you to be on the latest version of PowerShell.

## Adding snap-ins

Modules are the best practice in creating collections of cmdlets today for admins. However, many product teams use snap-ins and knowing how to load them is important.

```
Get-PSSnapin
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-PSSnapin" is run, and the output is displayed. The output shows one snap-in loaded:

```

PS C:\> Get-PSSnapin
Name      : Microsoft.PowerShell.Core
PSVersion : 4.0
Description : This windows Powershell snap-in contains cmdlets used to manage components of Windows Powershell.

PS C:\> -

```

Figure 11-2

The cmdlet **Get-PSSnapin** lists the currently loaded snap-ins. In our case, the core PowerShell snap-in is the only one listed. To see a list of registered and available snap-ins, use the **-Registered** parameter.

```
Get-PSSnapin -Registered
```

In our case, this didn't produce any additional results. However, if you tried this on a SQL server or Exchange server, you would see a list of snap-ins. Here's how to load a snap-in so you have access to the cmdlets.

```
Add-PSSnapin -Name *Exch*
```

The above command will load all the Exchange snap-ins into the shell. You can then find a list of all the cmdlets, by using the **Get-Command** cmdlet.

```
Get-Command -Module *Exch*
```

To remove a snap-in, use the **Remove-PSSnapin** cmdlet. However, many of us just close the shell.

## Importing modules

Most of the time, the collection of cmdlets you are looking for will be in a module. Here's how to see a list of all modules currently installed on your computer.

```
Get-Module -ListAvailable
```

The list produced represents many of the products you have installed, so if you have the RSAT tools for Windows 8 installed, you will see many more modules.

As we described earlier, starting with PowerShell version 3, you don't need to manually import the modules—you can just type the name of the cmdlet (or use **Get-Help** on the cmdlet) that you wish to use. If you want a list of all the cmdlets available in a module, do the following:

```
Get-Command -Module NetTCPIP
```

Administrator: Windows PowerShell		
PS C:\> Get-Command -Module NetTCPIP		
CommandType	Name	ModuleName
Function	Find-NetRoute	NetTCPIP
Function	Get-NetCompartment	NetTCPIP
Function	Get-NetIPAddress	NetTCPIP
Function	Get-NetIPConfiguration	NetTCPIP
Function	Get-NetIPInterface	NetTCPIP
Function	Get-NetIPv4Protocol	NetTCPIP
Function	Get-NetIPv6Protocol	NetTCPIP
Function	Get-NetNeighbor	NetTCPIP
Function	Get-NetOffloadGlobalSetting	NetTCPIP
Function	Get-NetPrefixPolicy	NetTCPIP
Function	Get-NetRoute	NetTCPIP
Function	Get-NetTCPConnection	NetTCPIP

Figure 11-3

As a reminder: when you attempt to run one of the cmdlets or use **Get-Help** on a cmdlet, the module will be automatically loaded.

## Sometimes you need to manually import modules

There will be times when you need to manually import a module. Later in this book, you will run across a few of these, such as when using PowerShell Remoting and connecting to a remote computer, sometimes you will need to manually import the module to gain access to a PSProvider that is loaded with the module. Also, if you write your own module (and you will, later) and the module is not stored in a special location, you will need to manually import the module.

The most common reason to import a module is one you can prevent from occurring. If you're using PowerShell version 2, you will need to manually import modules. Upgrade your management station to PowerShell version 4 to prevent this. Keep in mind, there will be times when you manage a server that it might still have PowerShell version 2, so you should know how to manually import modules.

To manually import a module, first find a list of the modules available on the system, by using **Get-Module -ListAvailable**, and then **Import-Module -Name <moduleName>** such as in the following example.

```
Import-Module -Name ActiveDirectory
```

```
Get-Command -Module ActiveDirectory
```

Administrator: Windows PowerShell		
CommandType	Name	ModuleName
Cmdlet	Add-ADCentralAccessPolicyMember	ActiveDir...
Cmdlet	Add-ADComputerServiceAccount	ActiveDir...
Cmdlet	Add-ADDomainControllerPasswordReplicationPolicy	ActiveDir...
Cmdlet	Add-ADFineGrainedPasswordPolicySubject	ActiveDir...
Cmdlet	Add-ADGroupMember	ActiveDir...
Cmdlet	Add-ADPrincipalGroupMembership	ActiveDir...
Cmdlet	Add-ADResourcePropertyListMember	ActiveDir...
Cmdlet	Clear-ADAccountExpiration	ActiveDir...
Cmdlet	Clear-ADClaimTransformLink	ActiveDir...
Cmdlet	Disable-ADAccount	ActiveDir...
Cmdlet	Disable-ADOptionalFeature	ActiveDir...

Figure 11-4

That's all there is to it! You can now extend PowerShell to manage other products!

# Exercise 11 – Extending PowerShell

**Time: 20 Minutes**

The goal of this exercise is to discover a new module, load it into memory, and run some of the commands that it adds to the shell. This is a great review of the techniques you have learned throughout the book to discover new commands and learn how to use them.

## Task 1

Discover a module that will let you work with TCP/IP.

## Task 2

PowerShell version 4 will dynamically load this module, but can you manually import the module?

## Task 3

Find a cmdlet that will list your computer's IP address. Can you determine how to add another IP address to your computer?

## Task 4

Take a few minutes and discover cmdlets that you might want to use in the future, by using the **Get-Help** cmdlet. Try to look for things like Schedule, Volume, and DNS. Keep in mind that some of these you will not find if the product is not installed. Discover these cmdlets and try to run them. Notice that the modules will automatically load. You can check using **Get-Module**.



## Chapter 12

# Storing data for later

### Learning the ropes

As your one-liners grow in length, it becomes easier to break those one-liners into multiple lines and start scripting. Often, you will need to store results of one command until you're ready to use it in another command, later. Like any good scripting environment, PowerShell supports the use of variables. There's no strict difference in functionality between using the shell interactively and writing a script. Therefore, you can use variables interactively, as well as in your scripts.

Variables play a key role in PowerShell, as they do in most scripting technologies. A variable is a placeholder for some value. The value of the placeholder might change based on script actions or intentional changes. In other words, the value is variable.

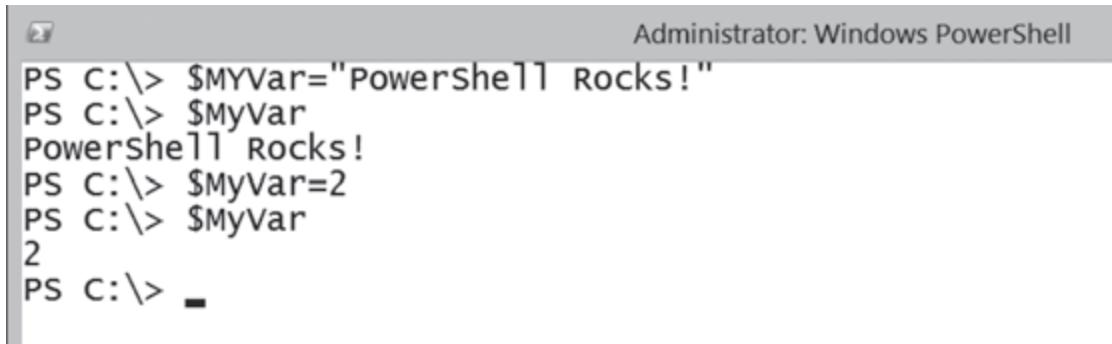
### Defining variables in PowerShell

PowerShell variable names must begin with \$. Take note that the \$ is not actually part of the variable name, just a way for PowerShell to recognize that you are using one.

```
$MyVar="PowerShell Rocks!"  
$MyVar=2
```

You can see what's stored in a variable in several different ways, below are the two most common. First, just type the name of the variable and press **Enter**, or use **Write-Output**.

```
$MyVar
Write-Output $MyVar
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$MYVar="PowerShell Rocks!" is entered, followed by PS C:\> \$MyVar, which outputs "PowerShell Rocks!". Then, PS C:\> \$MyVar=2 is entered, followed by PS C:\> \$MyVar, which outputs "2". The window has a standard Windows title bar and a dark grey background.

```
PS C:\> $MYVar="PowerShell Rocks!"
PS C:\> $MyVar
PowerShell Rocks!
PS C:\> $MyVar=2
PS C:\> $MyVar
2
PS C:\> -
```

Figure 12-1

In this example, we created a variable, **\$MyVar**, with a value of “PowerShell Rocks!” We can display the value of the variable by invoking the variable name. This variable will maintain this value until we close the shell or set **\$MyVar** to something else. We did in fact change the variable from a string to an integer of 2. How can you be sure that it changed from a string to an integer? Pipe the variable to **Get-Member**.

```
$MyVar | GM
```

There are two important things to note in this example:

1. We never had to formally declare the variable.
2. We’re not writing a script.

PowerShell allows variables to be used within the shell, or interactively, without requiring you to write a script. Variables used interactively stay in memory for the duration of the PowerShell session.

We can also set a variable to hold the results of a cmdlet.

```
$MyVar=Get-Service -name bits
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$Myvar=Get-Service -name bits is entered, followed by PS C:\> \$MyVar, which outputs a table:

Status	Name	DisplayName
Stopped	bits	Background Intelligent Transfer ser...

PS C:\>

Figure 12-2

In Figure 12-2, we create a variable called **\$MyVar**. The value of this variable is created by taking the object output of a **Get-Service** cmdlet. When you type **\$MyVar** at the next prompt, PowerShell displays the variable's contents in a formatted output.

PowerShell does not reevaluate the value of **\$MyVar** every time you invoke it. However, the value of **\$MyVar** is more than a collection of strings. In fact, it is an object that can be further manipulated.

```
$MyVar=Get-Service -name bits
$MyVar.status
```

```
$MyVar.Start()
$MyVar.status
```

```
$MyVar.Refresh()
$MyVar.status
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The commands entered are:

```
PS C:\> $Myvar=Get-Service -name bits
PS C:\> $MyVar.status
Stopped
PS C:\> $MyVar.Start()
PS C:\> $MyVar.status
Running
PS C:\> $MyVar.Refresh()
PS C:\> $MyVar.status
Running
PS C:\>
```

Figure 12-3

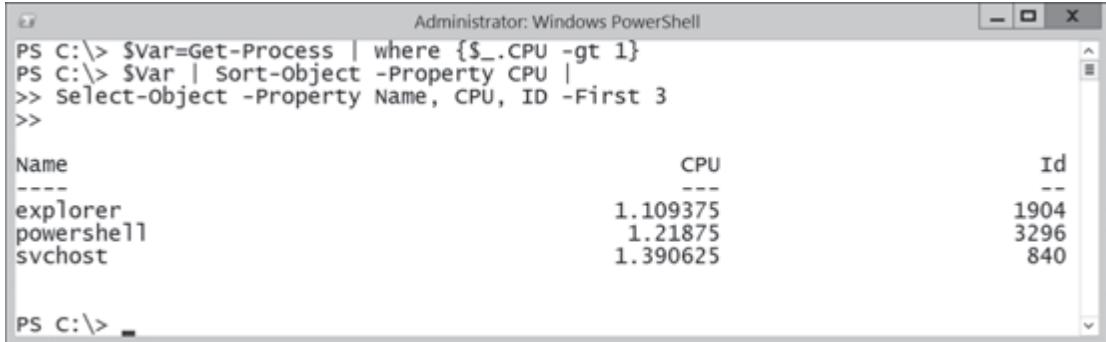
In Figure 12-3, we checked the status of the **bits** service. How did we know about this property? You can always pipe the variable to **Get-Member**, just as you did with other cmdlets. The status of the service was **Stopped**, so we executed a method on the **Object** to start the service (**\$MyVar.Start()**). We could have easily piped this to **Start-Service**. Notice that when we checked the status, it still reported **Stopped**. That's because the variable has been updated with the latest state, so we executed the **Refresh()** method. This updated the state, and then displayed the correct state.

## What about scripting?

Just as a note while we explore variables, everything we do in this chapter will work the same in a script. That's the beauty of PowerShell—what you can do in the shell works in the script. The following is a brief example.

```
$Var = Get-Process | where { $_.CPU -gt 1 }
$Var | Sort-Object -Property CPU | Select-Object -Property Name, CPU, ID -First 3
```

This command checks my computer for any processes with a CPU that is greater than 1, sorts the results by CPU, and then selects the first three results. Notice that we used the pipeline character as line continuation to make the command more readable.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> $var=Get-Process | where {$_._CPU -gt 1}
PS C:\> $var | Sort-Object -Property CPU |
>> Select-Object -Property Name, CPU, ID -First 3
>>
```

The output displays the following table:

Name	CPU	ID
explorer	1.109375	1904
powershell	1.21875	3296
svchost	1.390625	840

Figure 12-4

This same command produces the same results in our script.

The screenshot shows the PrimalScript 2014 interface. The title bar reads "PrimalScript 2014 - C:\Users\jasonnh\Documents\SAPIEN\Scripts\Untitled.ps1\*". The menu bar includes File, Home, View, Project, Deploy, Tools, and Help. The toolbar has icons for Paste, Select All, Find in Files..., Replace in Files..., Go to line, Platform, Build and Run, and Debug. The main editor window displays the following PowerShell script:

```

1 $Var=Get-Process | where {$_.CPU -gt 1}
2 $Var | Sort-Object -Property CPU |
3 |Select-object -Property Name, CPU, ID -First 3
4

```

The status bar at the bottom shows "Ready", "UNICODE LE", "CAP NUM Line 3, Col 1", and a small icon.

Figure 12-5

Using variables and commands can help break long one-liners into more bite-size pieces. We have a lot more scripting to do in future chapters, but you can see how easy it will be to transition from the shell to your script.

## Creating variables with cmdlets

So far, creating variables has been simple and easy in both the shell and your script. Specify the variable name and include a \$ sign, plus set it equal to something like the results of a command.

PowerShell also includes several cmdlets for working with variables and although we rarely use them, you will run across these and should know a little about them. You can find a list of the cmdlets by using **Get-Help**.

```
Get-Help *Variable*
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-Help \*variable\*". The output is a table listing cmdlets and provider cmdlets related to variables. The columns are Name, Category, Module, and Synopsis.

Name	Category	Module	Synopsis
Clear-Variable	Cmdlet	Microsoft.PowerShell.U...	Delete...
Get-Variable	Cmdlet	Microsoft.PowerShell.U...	Gets t...
New-Variable	Cmdlet	Microsoft.PowerShell.U...	Create...
Remove-Variable	Cmdlet	Microsoft.PowerShell.U...	Delete...
Set-Variable	Cmdlet	Microsoft.PowerShell.U...	Sets t...
Variable	Provider	Microsoft.PowerShell.Core	Provid...
about_Automatic_Variables	HelpFile		SHORT ...
about_Environment_Variables	HelpFile		SHORT ...
about_Preference_Variables	HelpFile		SHORT ...
about_Remote_Variables	HelpFile		SHORT ...
about_Variables	HelpFile		SHORT ...

Figure 12-6

You can probably guess at what most of these cmdlets do and what the additional `About_Help` files provide, but let's examine a few.

## Get-Variable

Recall that you can find a variable's value by typing out the variable name. But what if you forgot the variable name? If you can remember at least part of it, you can use **Get-Variable** to list all matching variables and their values.

```
Get-Variable -Name v*
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-Variable -Name v\*". The output lists variables whose names contain "v", including "Var", "var1", "var2", "var3", and "VerbosePreference". For each variable, the value is displayed.

Name	value
Var	{System.Diagnostics.Process (explorer), System...
var1	3
var2	Hello
var3	{ADWS, AeLookupSvc, ALG, AppHostSvc...}
VerbosePreference	SilentlyContinue

Figure 12-7

In Figure 12-7, we are finding all the variables that begin with the letter “v”. Notice that we didn’t need to use `$v*`. The \$ symbol is used in conjunction with the variable name when used in the shell.

Run `Get-Variable -Name *` to see all the defined variables.

```
Get-Variable -Name *
```

Name	value
---	---
\$	cls
?	True
^	cls
args	{}
ConfirmPreference	High
ConsoleFileName	SilentlyContinue
DebugPreference	{The term 'Var1=3' is not recognized as the n...
Error	Continue
ErrorActionPreference	NormalView
ErrorView	System.Management.Automation.EngineIntrinsics
ExecutionContext	False
false	4
FormatEnumerationLimit	

Figure 12-8

The output has been truncated, but you will recognize some of these variables from our earlier examples. However, note that variables such as `$MaximumErrorCount` or `$PSHome` are PowerShell’s automatic variables that are set by the shell. Read the `About_Automatic_Variables` Help for more information about what each one of these represents.

## Set-Variable/New-Variable

The original PowerShell cmdlet for creating variables is `Set-Variable`, which has an alias of `Set`. However, `New-Variable` was added because of the more appropriate verb name. They can be used interchangeably, and the best practice would be to use `New-Variable`, however you should recognize it as you will see this often in older scripts. The syntax for `Set-Variable` is as follows:

```
Set-Variable var "Computername"
```

This is the same as typing `$var="Computername"`. This cmdlet has several parameters that you might find useful. For one thing, you can define a read-only, or constant, variable.

```
New-Variable -Option "Constant" -Name PI -Value 3.1416
Set-Variable -name PI -Value "PowerShell"
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered was "Set-Variable -name PI -Value \"PowerShell\"". The output shows an error message: "Set-Variable : Cannot overwrite variable PI because it is read-only or constant." Below the error message, a detailed stack trace is provided, starting with "At line:1 char:1" and ending with "s.SetVariableCommand". The PowerShell prompt "PS C:\>" is visible at the bottom.

```

Administrator: Windows PowerShell
PS C:\> New-Variable -Option "Constant" -Name PI -Value 3.1416
PS C:\> Set-Variable -name PI -Value "PowerShell"
Set-Variable : Cannot overwrite variable PI because it is read-only or
constant.
At line:1 char:1
+ Set-Variable -name PI -Value "PowerShell"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo          : WriteError: (PI:String) [Set-Variable], Session
StateUnauthorizedAccessException
+ FullyQualifiedErrorId : VariableNotWritable,Microsoft.PowerShell.Command
s.SetVariableCommand
PS C:\>

```

Figure 12-9

By using the **-Option** parameter, we specified that we wanted the variable to be a constant. Once set, you cannot change the value or clear or remove the variable. It will exist for as long as your PowerShell session is running. If you close the shell and reopen it, the constant no longer exists.

If you need to change the variable value, use **Set-Variable**. However, even that will not work in this example, because \$PI was created as a constant.

You might wonder how you will know when different cmdlets should be used. The answer is that it probably depends on what type of variables you are creating and how they will be used. For example, you may want to use **New-Variable** to create all the empty variables you will need at the beginning of a script, and then use **Set-Variable** to define them. Using different cmdlets may help you keep track of what is happening to a variable throughout the script.

## Clear-Variable

You probably can figure out what **Clear-Variable** does without looking at the Help file or us explaining. Yup, it clears the current value of the variable.

```
Clear-Variable var
```

Notice that we didn't need to use **\$var**, just **var**. We also could have used **\$var=""**.

Technically, using **\$var=""** is not the same thing as using the **Clear-Variable** cmdlet. The cmdlet actually erases the value. The expression **\$var=""** is really setting the value of **\$var** to a string object with a length of 0. In most instances, this shouldn't be an issue, but if in doubt, use **Clear-Variable**.

You can also clear multiple variables with a single command.

```
$var = 1
$var2 = 2
$var3 = 3
Clear-Variable var*
```

In this example, we created variables `$var`, `$var2`, and `$var3`, followed by **Clear-Variable var\*** to clear the values of any variables that started with `var`.

## Remove-Variable

When you want to remove the variable and its value, use the **Remove-Variable** cmdlet. The syntax is essentially the same as **Clear-Variable**. You can remove a single variable.

```
$var = "foobar"
Remove-Variable var
```

Alternatively, you can remove multiple variables by using a wildcard.

```
Remove-Variable var*
```

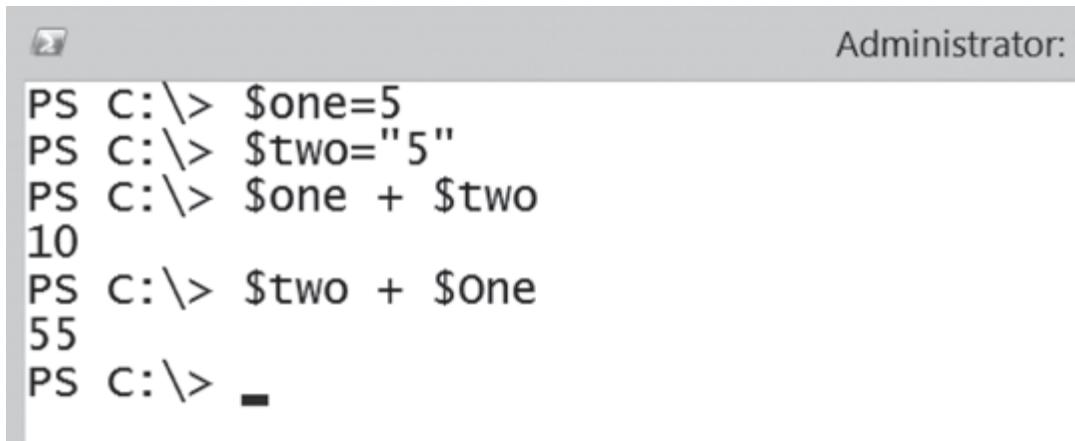
Of course, closing the shell (or when the script exits) will also remove the variable. You can see that the cmdlets for variables provide some additional options if you need them. However, we usually stick to the simple method that we described in the beginning of this chapter.

## Forcing a data type

The fact that PowerShell is built on and around .NET gives PowerShell tremendous power, which isn't always obvious. For example, in the first chapter, we explained that any PowerShell variable can contain any type of data. This occurs because all types of data—strings, integers, dates, etc.—are .NET classes that inherit from the base class named `Object`. A PowerShell variable can contain anything that inherits from `Object`. However, PowerShell can certainly tell the difference between different classes that inherit from `Object`.

You can force PowerShell to treat objects as a more specific type. For example, take a look at the sequence shown in Figure 12-10.

```
$one = 5
$two = "5"
$one + $two
$two + $one
```



The screenshot shows a Windows PowerShell window titled "Administrator:". The command PS C:\> \$one=5 is entered, followed by PS C:\> \$two="5". Then, PS C:\> \$one + \$two is run, resulting in 10. Next, PS C:\> \$two + \$one is run, resulting in 55. Finally, PS C:\> - is run.

```

PS C:\> $one=5
PS C:\> $two="5"
PS C:\> $one + $two
10
PS C:\> $two + $one
55
PS C:\> -

```

Figure 12-10

In this example, we gave PowerShell two variables: one contained the number five and the other contained the string character “5”. Even though this might look the same to you, it’s a big difference to a computer! However, we didn’t specify what type of data they were, so PowerShell assumed they were both of the generic object type. PowerShell also decided it would figure out something more specific when the variables are actually used.

When we added **\$one** and **\$two**, or **5 + “5”**, PowerShell said, “Aha, this is addition: The first character is definitely not a string because it wasn’t in double quotes. The second character one was in double quotes, but...well, if I take the quotes away it looks like a number, so I’ll add them.” This is why we correctly received ten as the result.

However, when we added **\$two** and **\$one**—reversing the order—PowerShell had a different decision to make. This time PowerShell said, “I see addition, but this first operand is clearly a string. The second operand is a generic object. So, let’s treat it like a string too and concatenate the two.” This is how we received 55 as the result, which is the first five tacked onto the second five.

But what about this example?

```
[int]$two + $one
```

It’s the same order as the example that resulted in “55”, but in this type we specifically told PowerShell that the generic object in **\$two** was an **[Int]**, or integer, which is a type PowerShell knows about. So, this time PowerShell used the same logic as in the first example. When it added the two, the result was 10.

You can force PowerShell to treat anything as a specific type.

```
$int = [int]"5"
[int] $int
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$int=[int]"5" is entered, followed by PS C:\> \$int | gm. The output shows the TypeName as System.Int32 and a detailed list of methods for the System.Int32 type.

Name	MemberType	Definition
CompareTo	Method	int CompareTo(system.Object value), int CompareTo(int...)
Equals	Method	bool Equals(system.Object obj), bool Equals(int obj),...
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCode GetTypeCode(), System.TypeCode IConve...
ToBoolean	Method	bool IConvertible.ToBoolean(System.IFormatProvider pr...
ToByte	Method	byte IConvertible.ToByte(System.IFormatProvider provi...
ToChar	Method	char IConvertible.ToChar(System.IFormatProvider provi...
ToDateTime	Method	datetime IConvertible.ToDateTime(System.IFormatProvid...

Figure 12-11

Here, the value “5” would normally be either a string object or, at best, a generic object. However, by specifying the type **[int]**, we forced PowerShell to try and convert “5” into an integer before storing it in the variable **\$int**. The conversion was successful, which you can see when we piped **\$int** to **Get-Member**, revealing the object’s type: **System.Int32**.

Note that once you apply a specific type to a variable, it stays that way until you specifically change it.

```
[int]$int = 1
```

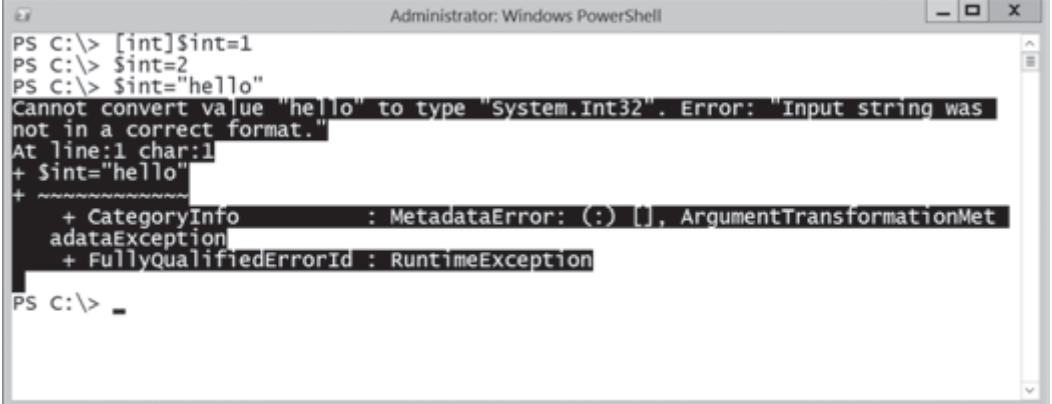
This creates a variable named **\$int** as an integer and assigns it the value 1. The **\$int** variable will be treated as an integer from now on, even if you don’t include the type.

```
$int = 2
```

It is still using **\$int** as an integer because it was already cast into a specific type. Once set up to be an integer, you can’t put other types of data into it. Here’s an example of an error that occurred when we tried to put a string into a variable that was already specifically cast as an integer.

```
[int]$int = 1
[int]$int = 2
[int]$int = "hello"
```

You will receive an error: Cannot convert value “hello” to type “System.Int32”. Error: “Input string was not in a correct format.”



```
Administrator: Windows PowerShell
PS C:\> [int]$int=1
PS C:\> $int=2
PS C:\> $int="hello"
Cannot convert value "hello" to type "System.Int32". Error: "Input string was
not in a correct format."
At line:1 char:1
+ $int="hello"
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMet
adataException
+ FullyQualifiedErrorId : RuntimeException
PS C:\> _
```

Figure 12-12

However, you can recast a variable by reassigning a new, specific type.

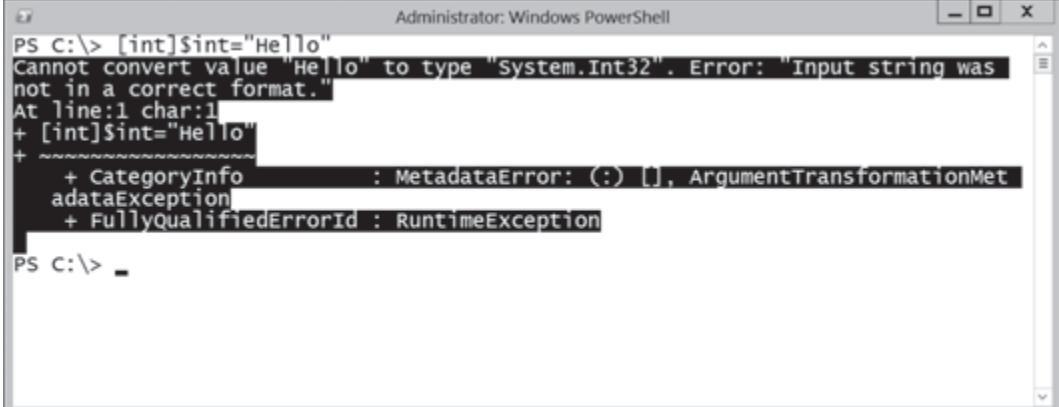
```
[string]$int = "Hello"
```

That works just fine, and **\$int** will now be treated as a string by PowerShell.

PowerShell isn't a miracle worker—for example, if you try to force it to convert something that doesn't make sense, it will complain.

```
[int]$int = "Hello"
```

Again, another error: Cannot convert “Hello” to “System.Int32”. Error: “Input string was not in a correct format.”



```
Administrator: Windows PowerShell
PS C:\> [int]$int="Hello"
Cannot convert value "Hello" to type "System.Int32". Error: "Input string was
not in a correct format."
At line:1 char:1
+ [int]$int="Hello"
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMet
adataException
+ FullyQualifiedErrorId : RuntimeException
PS C:\> _
```

Figure 12-13

This occurred because “Hello” can’t sensibly be made into a number.

This demonstrates not only that variables are objects, but also that PowerShell understands different types of data and provides different capabilities for the different types of data.

Curious about what object types are available? Here’s a quick list of more common types (although there are more than this).

- Array
- Bool (Boolean)
- Byte
- Char (a single character)
- Char[] (Character array)
- Decimal
- Double
- Float
- Int (Integer)
- Int[] (Integer array)
- Long (Long integer)
- Long[] (Long integer array)
- Regex (Regular expression)
- Scriptblock
- Single
- String
- XML

You can learn more about the variety of data types supported by .NET on [MSDN.Microsoft.com](https://msdn.microsoft.com).

## Variable with multiple objects (arrays)

An array is essentially a container for storing things. For almost all administrative scripts, simple arrays will suffice. When you think about a simple array, picture an egg carton with a single row. You can make the egg carton as long as you want. With a simple array, you can put numbers, strings, or objects in each compartment. Multi-dimensional arrays exist, but they are beyond the scope of what we will cover here.

### Array or collection

When reading technical material, you may also come across the term Collection, which is a type of array that is usually created as the result of some query. For example, you might execute a query to return all instances of logical drives on a remote server using Windows Management Instrumentation (WMI).

The resulting object will hold information about all logical drives in a collection. This collection object is handled in the much same way as an array when it comes time to enumerate the contents. For now, remember that when you see the term collection, think array. Or vice-versa—it doesn't really matter which term you use.

You can create an array by defining a variable and specifying the contents as delimited values.

```
$a = 1, 2, 3, 4, 5, 6, 7
```

To view the contents of \$a, all you need to do is type \$a.

```
$a
```

The screenshot shows a Windows PowerShell window with the title bar 'Administrator: W'. The command PS C:\> \$a=1,2,3,4,5,6,7 is entered, followed by PS C:\> \$a. The output shows the numbers 1 through 7 on separate lines. The prompt PS C:\> is visible at the bottom.

Figure 12-14

If you want to create an array with a range of numbers, you should use the range operator (..).

```
$a = 2..7
$a
```

To create an array with strings, you must enclose each element in quotes.

```
$servers = "dc1", "app02", "print1", "file3"
```

PowerShell arrays can also contain objects.

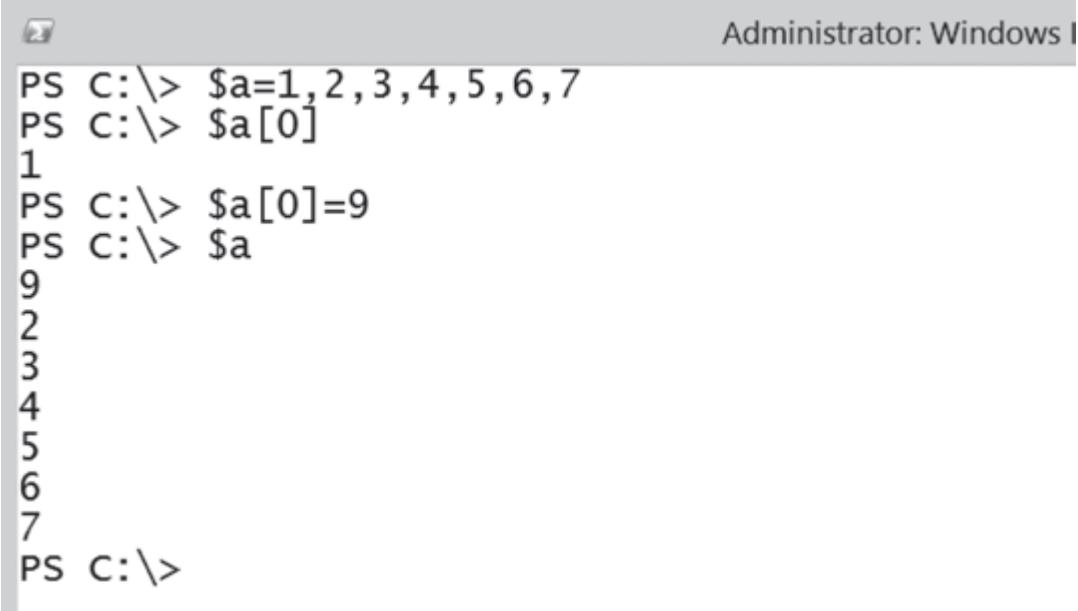
```
$svc = Get-Service | where { $_.status -eq "running" }
```

We've created an array called **\$svc** that contains all running services.

## Working with arrays

When you work with individual elements in an array, the first important thing to remember is that arrays start counting at 0. To reference a specific element, the syntax is **\$arrayname[index]**.

```
$a = 1, 2, 3, 4, 5, 6, 7
$a[0]
$a[0] = 9
$a
```



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command '\$a = 1, 2, 3, 4, 5, 6, 7' is entered and executed, displaying the array [1, 2, 3, 4, 5, 6, 7]. Then, '\$a[0]' is entered and executed, displaying the value 1. Next, '\$a[0] = 9' is entered and executed, changing the first element to 9. Finally, '\$a' is entered and executed, displaying the modified array [9, 2, 3, 4, 5, 6, 7]. The prompt 'PS C:\>' appears at the bottom.

```
PS C:\> $a=1,2,3,4,5,6,7
PS C:\> $a[0]
1
PS C:\> $a[0]=9
PS C:\> $a
9
2
3
4
5
6
7
PS C:\>
```

Figure 12-15

Not only did we display a single element of the array—element 0—but we also changed that element to the number 9.

It's also easy to create an array from an existing source, instead of manually typing a list of names. If you already have a list of server names, such as Servers.txt, you can read that file with the **Get-Content** cmdlet, and then populate the array.

```
$Serverlist = Get-Content -Path C:\Servers.txt
Get-Service -name bits -ComputerName $Serverlist | Select name, MachineName
```

```
Administrator: Windows PowerShell
PS C:\> $Serverlist=Get-Content -Path C:\Servers.txt
PS C:\> Get-Service -name bits -ComputerName $Serverlist | select name, MachineName
Name          MachineName
---          -----
bits          Localhost
bits          SapienDC

PS C:\> _
```

Figure 12-16

In Figure 12-16, Servers.txt is a simple text file with a list of computer names. We create the array **\$serverlist** by invoking **Get-Content**. Invoking **\$serverlist** merely displays the array's contents. Armed with this array, you can use the variable in cmdlets.

## Associative arrays

Associative arrays are special types of arrays. Think of them as a way of relating, or associating, one piece of data with another piece of data. For example, they're useful for performing certain types of data lookup that we'll see in this chapter. You may also see associative arrays referred to as dictionaries or hash tables.

An associative array is a data structure that stores multiple key-value pairs.

<u>Key</u>	<u>Value</u>
BMW	Blue
Honda	Red
Ford	Blue

The key is usually some piece of well-known or easily obtained information, while the value is the data you want to look up. Keys are unique in the array, while values can be repeated as necessary. In our table example, the associative array keys are car names, while each corresponding value is the car's color. Notice that the key names are unique while data values can be duplicated.

PowerShell uses the hash table data type to store the contents of an associative array because that data type provides fast performance when looking up data. What's really neat about associative arrays is that the individual values can be of different types. In other words, one key might be connected to a string, while another is connected to an XML document. This lets you store any kind of arbitrary data you want in an array.

## Creating an associative array

The `@` operator is used to create an associative array.

```
$cars = @{ "BMW" = "Blue"; "Honda" = "Red"; "Ford" = "Blue" }
```

The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command PS C:\> \$cars=@{"BMW"="Blue";"Honda"="Red";"Ford"="Blue"} is entered, followed by PS C:\> \$cars. The output is a table:

Name	Value
Honda	Red
BMW	Blue
Ford	Blue

Figure 12-17

This creates an associative array with three keys and three values, exactly as shown in the previous table. Of course, the values don't have to be simple strings. They can be numbers, the output from a cmdlet, or even other associative arrays.

By the way, you can also use the `@` operator to create a normal array.

```
$a = @(1, 2, 3, 4, 5)
```

The difference is that only values were given to the `@` operator; we didn't give it key=value pairs, so it created a normal one-dimensional array for us. Also associative arrays use curly braces {} and regular arrays use parentheses () .

## Using an associative array

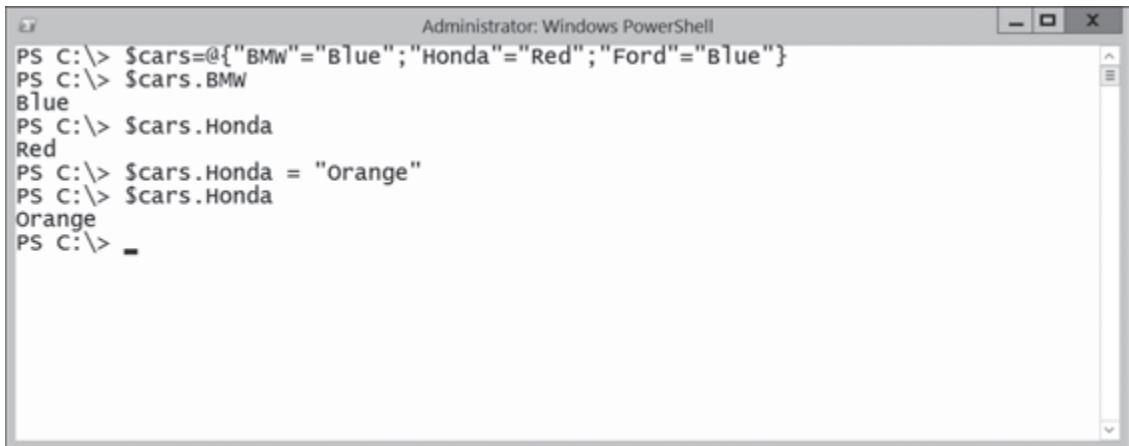
You can display the entire contents of an associative array by calling its name.

```
$cars = @{ "BMW" = "Blue"; "Honda" = "Red"; "Ford" = "Blue" }
$cars
```

However, normally you'd want to access a single element.

```
$cars = @{ "BMW" = "Blue"; "Honda" = "Red"; "Ford" = "Blue" }
```

```
$cars.BMW
$cars.Honda
$cars.Honda = "Orange"
$cars.Honda
```



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command PS C:\> \$cars=@{"BMW"="Blue";"Honda"="Red";"Ford"="Blue"} is entered and executed. Then, PS C:\> \$cars.BMW is entered and the value 'Blue' is displayed. Next, PS C:\> \$cars.Honda is entered and the value 'Red' is displayed. Finally, PS C:\> \$cars.Honda = "Orange" is entered and executed, followed by PS C:\> \$cars.Honda which displays the updated value 'Orange'.

Figure 12-18

Type the associative array variable name, a period, and then the key you want to retrieve. PowerShell will respond by displaying the associated value. You can also change the value of the key, as we did above.

## Handling variables inside of quotes

An often confusing topic for people starting out with PowerShell is the difference between single quotes (') and double quotes (""). PowerShell treats these differently. Let's spend a few minutes looking at them.

We'll start with an example of using a variable inside of a double quote, then single quotes. You will probably see the difference right away.

```
$Myvar = "Cool"
"PowerShell is really $MyVar"
'PowerShell is really $MyVar'
```

Notice that when you use double quotes, variables are replaced with the value stored in them. If you use single quotes, they are not.

The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. It contains the following command and its output:

```
PS C:\> $Myvar="Cool"
PS C:\> "Powershell is really $MyVar"
Powershell is really Cool
PS C:\> 'Powershell is really $MyVar'
Powershell is really $MyVar
PS C:\> "Powershell is really $MyVar - and the data was stored in $MyVar"
Powershell is really cool - and the data was stored in Cool
PS C:\> "Powershell is really $MyVar - and the data was stored in `"$MyVar"
Powershell is really cool - and the data was stored in $MyVar
PS C:\>
```

Figure 12-19

There are times when you want a variable replaced by its value and times when you don't. Look at the following example.

```
PS C:\> "PowerShell is really $MyVar - and the data was stored in $MyVar"
PowerShell is really Cool - and the data was stored in Cool
PS C:\> "PowerShell is really $MyVar - and the data was stored in `"$MyVar"
PowerShell is really Cool - and the data was stored in $MyVar
```

We really only wanted the value of the variable the first time—we didn't want it the second time. To stop PowerShell from replacing a variable in double quotes with its value, use the back-tick (`) character.

## Subexpressions

Inside of double quotes, PowerShell supports running expressions—subexpressions—that can provide a lot of flexibility.

```
"Add two numbers $(2 + 4)"
```

This adds two numbers and returns 6.

```
"The current bits service status: $(Get-service -name bits | Select status)"
```

This returns the result `@{ Status = Running }`.

The `$()` is recognized inside of double quotes as special. You can execute any expression you need, including cmdlets!

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history is as follows:

```

PS C:\> "Add two numbers $(2 + 4)"
Add two numbers 6
PS C:\> "The current bits service status: $(Get-service -name bits | Select stat us)"
The current bits service status: @{Status=Running}
PS C:\> $status=Get-service -name bits
PS C:\> "The current bits service status: $($status.status)"
The current bits service status: Running
PS C:\> "The current bits service status: $((Get-service -name bits).status)"
The current bits service status: Running
PS C:\> _

```

Figure 12-20

You can store results to a variable and use a subexpression to call the properties or methods of those results.

```
$status = Get-Service -Name bits
"The current bits service status: $($status.status)"
```

Or you can run the entire expression from inside the subexpression.

```
"The current bits service status: $((Get-Service -Name bits).status)"
```

### What's the point?

The point is to recognize the difference between single and double quotes. We prefer to use the quotes specifically as intended. For instance, if we want to store strings (like computer names) to an array, we will use single quotes, as there is no need to have the feature that double quotes provide.

```
$Var = 'server1', 'server2', 'server3'
```

However, when we write a script and expect to replace a variable with a value, we use double quotes, even if the cmdlet doesn't require it.

Storing data for later

```
$Var=Read-host "Enter the service name"  
Get-Service -Name "$Var"
```

You should take a few minutes and try the exercise so that you can play with variables, arrays, and quotes. You will need this information as we will be diving into scripting in the next chapter.

## Exercise 12 – Storing data

Time: 20 Minutes

In this exercise, you will store results into variables and retrieve those results when needed. Experiment on your own beyond this exercise, if you like.

### Task 1

Create a variable called **\$Services** and store all stopped services to the variable.

### Task 2

Using the answer from above, display only the third stopped service

### Task 3

Using **Read-Host** and a variable, prompt for a computer name, then retrieve the status of the **bits** service from that computer.

### Task 4

Store the newest 100 entries of the system log to a variable. Using the variable and a pipeline, display only the entries that have an entry **Type** of **Error**.

### Task 5

Using the command from the last task and the cmdlet **Write-Output**, display the first error message.

### Task 6

Using the command from task 4 and the cmdlet **Write-Output**, display the first error message in the color **Yellow**.

## Chapter 13

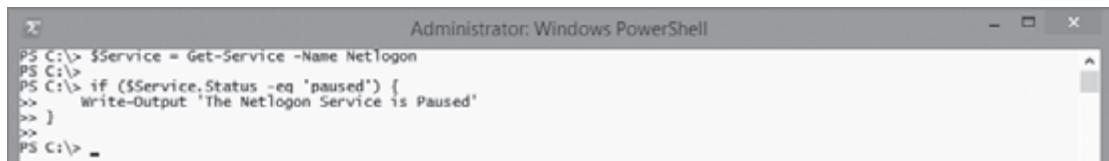
# Making your script smarter

### Making decisions (If)

The **If** statement executes a block of code if a specified conditional expression evaluates to **True**. This logic can also be reversed so that the block of code is executed if the specified conditional expression evaluates to **False**. For this reason, **If** is more of a decision-making construct than a looping one.

We'll write an **If** statement so that if the condition contained within the parentheses is **True** (if the Netlogon service is paused), then the code contained within the curly braces will be executed.

```
$Service = Get-Service -Name Netlogon  
  
if ($Service.Status -eq 'paused') {  
    Write-Output 'The Netlogon Service is Paused'  
}
```



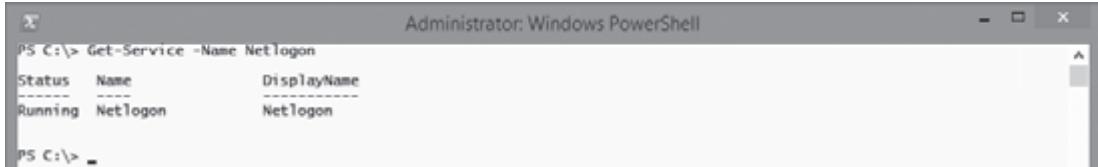
The screenshot shows an Administrator Windows PowerShell window. The title bar reads "Administrator: Windows PowerShell". The command line shows the execution of a PowerShell script. The script starts with \$Service = Get-Service -Name Netlogon, followed by an if statement where the condition is (\$Service.Status -eq 'paused'). Inside the if block, there is a Write-Output command with the message 'The Netlogon Service is Paused'. The script ends with a closing brace for the if block. The command PS C:\> is visible at the bottom of the window.

Figure 13-1

### Windows PowerShell: TFM

Notice that in Figure 13-1, nothing was executed. That's because the Netlogon service on our workstation is currently running.

```
Get-Service -Name Netlogon
```

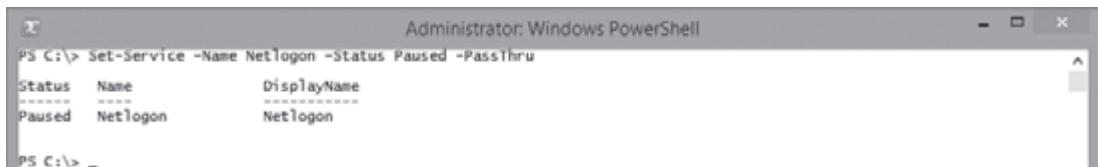


```
Administrator: Windows PowerShell
PS C:\> Get-Service -Name Netlogon
Status      Name        DisplayName
Running     Netlogon   NetLogon
PS C:\>
```

Figure 13-2

We'll set that particular service to **Paused** on our workstation.

```
Set-Service -Name Netlogon -Status Paused -PassThru
```

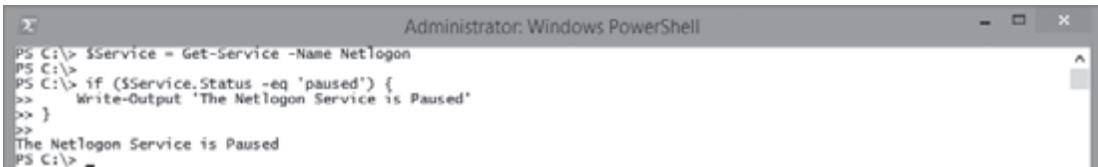


```
Administrator: Windows PowerShell
PS C:\> Set-Service -Name Netlogon -Status Paused -PassThru
Status      Name        DisplayName
Paused     Netlogon   NetLogon
PS C:\>
```

Figure 13-3

Now, we'll run the same **If** statement again and notice that the code within the curly braces does, indeed, execute this time because the condition that we tested for is now **True**.

```
$Service = Get-Service -Name Netlogon
if ($Service.Status -eq 'paused') {
    Write-Output 'The Netlogon Service is Paused'
}
```



```
Administrator: Windows PowerShell
PS C:\> $Service = Get-Service -Name Netlogon
PS C:\>
PS C:\> if ($Service.Status -eq 'paused') {
>>     Write-Output 'The Netlogon Service is Paused'
>> }
>> The Netlogon Service is Paused
PS C:\>
```

Figure 13-4

We set the Netlogon service back to **Running**.

```
Set-Service -Name Netlogon -Status Running -PassThru
```

Status	Name	DisplayName
Running	Netlogon	NetLogon

Figure 13-5

If we wanted the block of code contained within the curly braces to execute if the condition contained within the parentheses was **False**, then we would reverse the logic by using the **-not** operator.

```
$Service = Get-Service -Name Netlogon
if (-not ($Service.Status -eq 'paused')) {
    Write-Output 'The Netlogon Service is not Paused'
}
```

```
PS C:\> $Service = Get-Service -Name Netlogon
PS C:\>
PS C:\> if (-not ($Service.Status -eq 'paused')) {
>>     Write-Output 'The Netlogon Service is not Paused'
>> }
>>
The Netlogon Service is not Paused
PS C:\> _
```

Figure 13-6

While we don't use aliases in our code, it's good to know that the exclamation point is an alias for the **-not** operator, because you'll probably see others using it.

We can combine these two approaches and execute one block of code if a condition is true, and another if it is false, by adding an optional **Else** block to our first example.

The first condition in our **If** statement is evaluated and it evaluates to **False**, so the code within the first set of curly braces does not execute. The evaluation process continues and the **Else** block is like a catch all, so if nothing else evaluates to **True**, the code contained in the curly braces of the **Else** block will execute, as shown in the following example.

```
$Service = Get-Service -Name Netlogon

if ($Service.Status -eq 'paused') {
    Write-Output 'The Netlogon Service is Paused'
}
else {
    Write-Output 'The Status of the Netlogon Service is Unknown'
}
```

```
Administrator: Windows PowerShell
PS C:\> $Service = Get-Service -Name Netlogon
PS C:\>
PS C:\> if ($Service.Status -eq 'paused') {
>>     Write-Output 'The Netlogon Service is Paused'
>> }
>> else {
>>     Write-Output 'The Status of the Netlogon Service is Unknown'
>> }
>>
The Status of the Netlogon Service is Unknown
PS C:\> _
```

Figure 13-7

There's also an optional **ElseIf** operator that can be added to test if additional conditions are true. Multiple **ElseIf** statements can be used, as shown in Figure 13-8 where we've added additional logic to test for two more conditions.

```
$Service = Get-Service -Name Netlogon

if ($Service.Status -eq 'paused') {
    Write-Output 'The Netlogon Service is Paused'
}
elseif ($Service.Status -eq 'running') {
    Write-Output 'The Netlogon Service is Running'
}
elseif (-not ($Service.Status -eq 'running')) {
    Write-Output 'The Netlogon Service is not Running'
}
else {
    Write-Output 'The Status of the Netlogon Service is Unknown'
}
```

```
Administrator: Windows PowerShell
PS C:\> $Service = Get-Service -Name Netlogon
PS C:\>
PS C:\> if ($Service.Status -eq 'paused') {
>>     Write-Output 'The Netlogon Service is Paused'
>> }
>> elseif ($Service.Status -eq 'running') {
>>     Write-Output 'The Netlogon Service is Running'
>> }
>> elseif (-not ($Service.Status -eq 'running')) {
>>     Write-Output 'The Netlogon Service is not Running'
>> }
>> else {
>>     Write-Output 'The Status of the Netlogon Service is Unknown'
>> }
>>
The Netlogon Service is Running
PS C:\> _
```

Figure 13-8

Keep in mind that an **If** statement runs from top to bottom and once a condition is met, the **If** statement doesn't evaluate any of the remaining conditions. We'll add an additional **Elseif** statement before any of the others, to demonstrate this.

We've now added an **Elseif** condition, reversing the logic to test if the Netlogon service is not paused; that is, if the evaluation is **False**, the **-not** operator reverses from **False** to **True**.

```
$Service = Get-Service -Name Netlogon

if ($Service.Status -eq 'paused') {
    Write-Output 'The Netlogon Service is Paused'
}
elseif (-not ($Service.Status -eq 'paused')) {
    Write-Output 'The Netlogon Service is not Paused'
}
elseif ($Service.Status -eq 'running') {
    Write-Output 'The Netlogon Service is Running'
}
elseif (-not ($Service.Status -eq 'running')) {
    Write-Output 'The Netlogon Service is not Running'
}
else {
    Write-Output 'The Status of the Netlogon Service is Unknown'
}
```

```
Administrator: Windows PowerShell
PS C:\> $Service = Get-Service -Name Netlogon
PS C:\>
PS C:\> if ($Service.Status -eq 'paused') {
>>     Write-Output 'The Netlogon Service is Paused'
>> }
>> elseif (-not ($Service.Status -eq 'paused')) {
>>     Write-Output 'The Netlogon Service is not Paused'
>> }
>> elseif ($Service.Status -eq 'running') {
>>     Write-Output 'The Netlogon Service is Running'
>> }
>> elseif (-not ($Service.Status -eq 'running')) {
>>     Write-Output 'The Netlogon Service is not Running'
>> }
>> else {
>>     Write-Output 'The Status of the Netlogon Service is Unknown'
>> }
>>
The Netlogon Service is not Paused
PS C:\>
```

Figure 13-9

Do you see the problem with the logic in Figure 13-9? Although the Netlogon service is **Running**, that particular **Elseif** condition is never evaluated because the service is currently **Not Paused**. So the second condition in our **If** statement is met and the **If** statement exits at that point, never testing for the other **Elseif** conditions.

It's also possible to nest **If** statements inside of other **If** statements. We know that if our service is paused, it cannot be running—because those statuses are mutually exclusive. If the status is not paused, it could be in one of many statuses, so we'll nest our test for running within that **Elseif** statement.

```
$Service = Get-Service -Name Netlogon

if ($Service.Status -eq 'paused') {
    Write-Output 'The Netlogon Service is Paused'
```

```

}
else {
    Write-Output 'The Netlogon Service is not Paused'

    if ($Service.Status -eq 'running') {
        Write-Output 'The Netlogon Service is Running'
    }
    elseif (-not ($Service.Status -eq 'running')) {
        Write-Output 'The Netlogon Service is not Running'
    }
    else {
        Write-Output 'The Status of the Netlogon Service is Unknown'
    }
}

```

```

Administrator: Windows PowerShell
PS C:\> $Service = Get-Service -Name Netlogon
PS C:\>
PS C:\> if ($Service.Status -eq 'paused') {
>>     Write-Output 'The Netlogon Service is Paused'
>> }
>> else {
>>     Write-Output 'The Netlogon Service is not Paused'
>> }
>>     if ($Service.Status -eq 'running') {
>>         Write-Output 'The Netlogon Service is Running'
>>     }
>>     elseif (-not ($Service.Status -eq 'running')) {
>>         Write-Output 'The Netlogon Service is not Running'
>>     }
>>     else {
>>         Write-Output 'The Status of the Netlogon Service is Unknown'
>>     }
>> }
>>
The Netlogon Service is not Paused
The Netlogon Service is Running
PS C:\> -

```

Figure 13-10

As you can see in Figure 13-10, two conditions were met because of the way we nested two **If** statements. Nesting **If** statements can become difficult to troubleshoot if you experience logic problems. However, sometimes nesting them is necessary.

If you want to test for multiple conditions, as in Figure 13-10, a better approach to accomplish this task is by using what's called a **Switch** statement, which we'll be covering next.

## Making decisions better (**switch**)

If you find yourself needing to check for multiple conditions or otherwise creating a lengthy **If/Else** statement, then you should consider using PowerShell's **Switch** statement. The **Switch** statement acts like an **If** statement, with many **ElseIf** statements. If you have experience with VBScript or VB.Net, you'll recognize this construct as the **Select Case** statement in those languages, and if you happen to have previously used C#, a **Switch** statement in PowerShell is just like a **Switch** statement in C#.

We've converted the final example that we used in the **If** section of this chapter into a **Switch** statement, as shown in the following example.

```
$Service = Get-Service -Name Netlogon

switch ($Service.Status) {
    Paused { 'Paused' }
    { $_ -ne 'Paused' } { 'Not Paused' }
    Running { 'Running' }
    { $_ -ne 'Running' } { 'Not Running' }
    Default { 'Unknown' }
}
```

```
Administrator: Windows PowerShell
PS C:\> $Service = Get-Service -Name Netlogon
PS C:\>
PS C:\> switch ($Service.Status) {
>>     Paused { 'Paused' }
>>     { $_ -ne 'Paused' } { "Not Paused" }
>>     Running { 'Running' }
>>     { $_ -ne 'Running' } { "Not Running" }
>>     Default { 'Unknown' }
>> }
>>
Not Paused
Running
PS C:\>
```

Figure 13-11

As you can see in Figure 13-11, a **Switch** statement runs top to bottom, just as an **If** statement does, but by default it tests for every condition, even if a condition is met.

The **Break** keyword can be used to break out of the **Switch** statement after the first match, if you would like the **Switch** statement to work more like the **If** statement, as shown in Figure 13-12.

```
$Service = Get-Service -Name Netlogon

switch ($Service.Status) {
    Paused { 'Paused'; break }
    { $_ -ne 'Paused' } { 'Not Paused'; break }
    Running { 'Running'; break }
    { $_ -ne 'Running' } { 'Not Running'; break }
    Default { 'Unknown' }
}
```

```
Administrator: Windows PowerShell
PS C:\> $Service = Get-Service -Name Netlogon
PS C:\>
PS C:\> switch ($Service.Status) {
>>     Paused { 'Paused'; break }
>>     { $_ -ne 'Paused' } { 'Not Paused'; break }
>>     Running { 'Running'; break }
>>     { $_ -ne 'Running' } { 'Not Running'; break }
>>     Default { 'Unknown' }
>> }
>>
Not Paused
PS C:\>
```

Figure 13-12

The **Switch** statement also supports additional options, which are outlined in the following table.

**Table 13.1 Switch Statement Options**

<u>Switch Options</u>	<u>Option Description</u>
-casesensitive	If the match clause is a string, this switch option modifies it to be case-sensitive. If the variable to be evaluated is not a string, this option is ignored.
-exact	Indicates that if the match clause is a string, it must match exactly. Use of this parameter disables -wildcard and -regex. If the variable to be evaluated is not a string, this option is ignored.
-file	Takes input from a file rather than a statement. If multiple -File parameters are provided, it uses the last one. Each line of the file is read and passed through the Switch block.
-regex	Indicates that the match clause, if a string, is treated as a regular expression string. Use of this parameter disables -wildcard and -exact. If the variable to be evaluated is not a string, this option is ignored.
-wildcard	Indicates that if the match clause is a string, it is treated as a -wildcard string. Use of this parameter disables -regex and -exact. If the variable to be evaluated is not a string, this option is ignored.

The complete **Switch** syntax can be one of the following:

```
switch [-regex|-wildcard|-exact][-casesensitive] ( pipeline )
```

or:

```
switch [-regex|-wildcard|-exact][-casesensitive] -file filename {
    "string"|number|variable|{ expression } { statementlist }
    default { statementlist
}
```

## Using the Foreach loop

The **Foreach** construct, not to be confused with the **ForEach-Object** cmdlet, is designed to iterate through a collection of objects. We've found that the **Foreach** construct is the type of loop that we most often use in our scripts and functions, so we're covering it first.

There are many different scenarios where you might use the **Foreach** construct. We'll cover a couple of the more common ones so you'll be prepared whenever you need to use this construct.

The first scenario we'll cover is when a cmdlet doesn't natively support multiple items on one of its parameters, like the **-ComputerName** parameter of the **Get-WinEvent** cmdlet. It only supports a string and not an array or strings (one computer name, not a comma separated list of computer names).

We'll use a **Foreach** loop to iterate, or step though, each item in the **\$Computers** array or collection.

```
$Computers = 'DC01', 'SQL01', 'WEB01'

foreach ($Computer in $Computers) {
    Get-WinEvent -ComputerName $Computer -LogName System -MaxEvents 1
}
```

```
Administrator: Windows PowerShell
PS C:\> $Computers = 'DC01', 'SQL01', 'WEB01'
PS C:\>
PS C:\> foreach ($Computer in $Computers) {
>>     Get-WinEvent -ComputerName $Computer -LogName System -MaxEvents 1
>> }
>>

ProviderName: Service Control Manager
TimeCreated          Id LevelDisplayName Message
-----      -- -----
3/4/2014 8:30:27 AM 7036 Information   The Remote Registry service entered the stopped state.
3/4/2014 9:42:34 AM 7036 Information   The Software Protection service entered the stopped state.
3/4/2014 9:42:34 AM 7036 Information   The Software Protection service entered the stopped state.

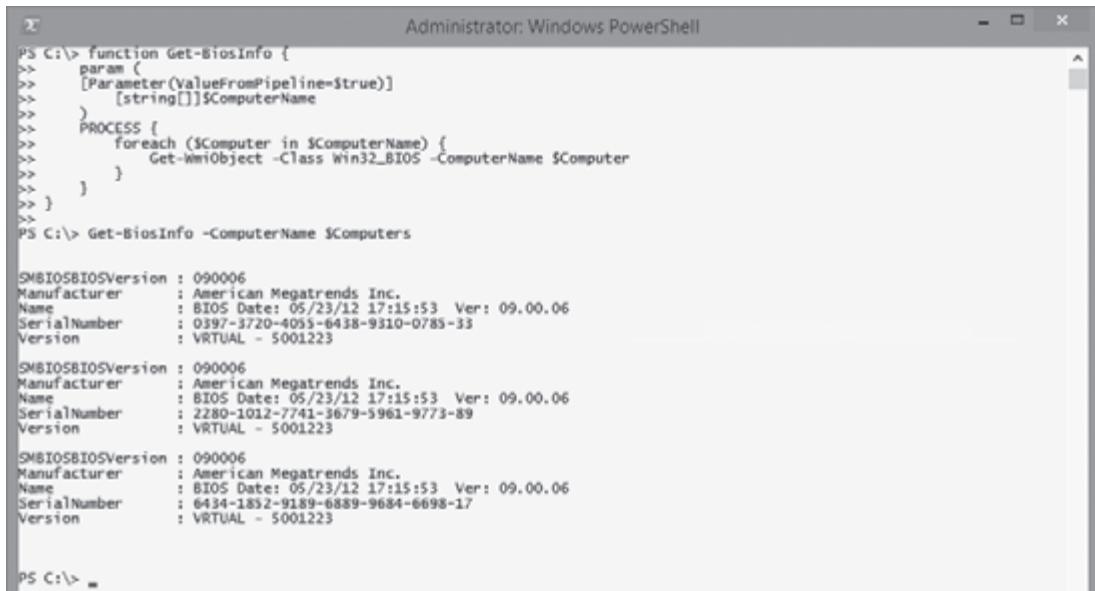
PS C:\> _
```

Figure 13-13

One of the things that was confusing to us, being non-developers, when we first started using the **Foreach** construct was where did the **\$Computer** variable come from? We can see where the **\$Computers** array or collection was defined, but what about **\$Computer**? It's a made-up variable—you could just as easily use the **\$\_** for the current object or **\$SantaClaus**, because it's just a made-up, temporary variable and PowerShell doesn't care what you call it as long as you use the same variable name inside the parentheses and curly braces.

One of the main places that you'll use the **Foreach** construct is in your functions and scripts that accept pipeline input. It's actually not needed when computer names are piped into the function, as shown in Figure 13-14, but it's needed when those computer names are sent in via parameter input to make the function work the same way, regardless of how the computer names are specified (pipeline versus parameter input).

```
function Get-BiosInfo {
    param (
        [Parameter(ValueFromPipeline=$true)]
        [string[]]$ComputerName
    )
    PROCESS {
        foreach ($Computer in $ComputerName) {
            Get-WmiObject -Class Win32_BIOS -ComputerName $Computer
        }
    }
}
```



```

Administrator: Windows PowerShell
PS C:\> function Get-BiosInfo {
>>     param (
>>         [Parameter(ValueFromPipeline=$true)]
>>         [string[]]$ComputerName
>>     )
>>     PROCESS {
>>         foreach ($Computer in $ComputerName) {
>>             Get-WmiObject -Class Win32_BIOS -ComputerName $Computer
>>         }
>>     }
>> }
PS C:\> Get-BiosInfo -ComputerName $Computers

SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 05/23/12 17:15:53 Ver: 09.00.06
SerialNumber      : 0397-3720-4093-6438-9310-0785-33
Version          : VIRTUAL - 5001223

SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 05/23/12 17:15:53 Ver: 09.00.06
SerialNumber      : 2280-1012-7742-3679-5961-9773-89
Version          : VIRTUAL - 5001223

SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 05/23/12 17:15:53 Ver: 09.00.06
SerialNumber      : 6434-1852-9189-6889-9684-6698-17
Version          : VIRTUAL - 5001223

PS C:\> =

```

Figure 13-14

The term **ForEach** is also an alias for the **ForEach-Object** cmdlet. We're pointing this out in case you find examples using the cmdlet, because that particular cmdlet alias works differently than the construct we've been demonstrating in this chapter. If you were to simply replace **ForEach** in the examples above with **ForEach-Object**, the examples wouldn't work.

Typically, you'll use the **ForEach** construct in a script or function and the **ForEach-Object** cmdlet in a one-liner when working at the PowerShell console.

## Using the For loop

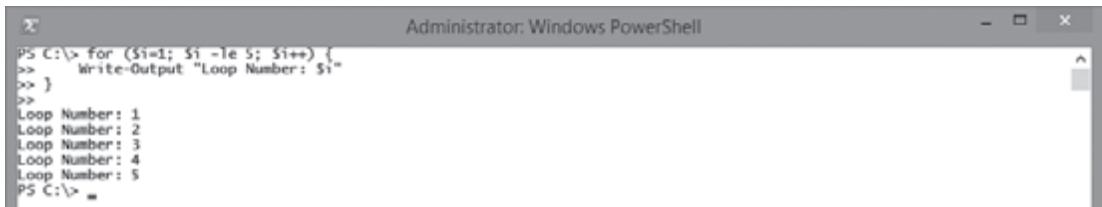
A **For** loop is similar to the **ForEach** loop. Also called the **For** statement, it will keep looping while some condition is met and it will execute a block of code with each iteration of the loop. The condition could be a counter. For example, we might need a loop that says, "Start counting at one and execute some block of code during each loop until you reach five." Or we might need a loop that says, "As long as some statement is true, keep looping and execute a block of code in each loop."

If you've used the **For** loop in VBScript, conceptually PowerShell's implementation is the same. However, the syntax of PowerShell's **For** loop may be confusing at first.

```

for ($i=1; $i -le 5; $i++) {
    Write-Output "Loop Number: $i"
}

```



```
Administrator: Windows PowerShell
PS C:\> for ($i=1; $i -le 5; $i++) {
>>     Write-Output "Loop Number: $i"
>> }
>>
Loop Number: 1
Loop Number: 2
Loop Number: 3
Loop Number: 4
Loop Number: 5
PS C:\> =
```

Figure 13-15

This syntax essentially instructs PowerShell that for (some set of conditions) {do this block of commands}.

## Using the While loop

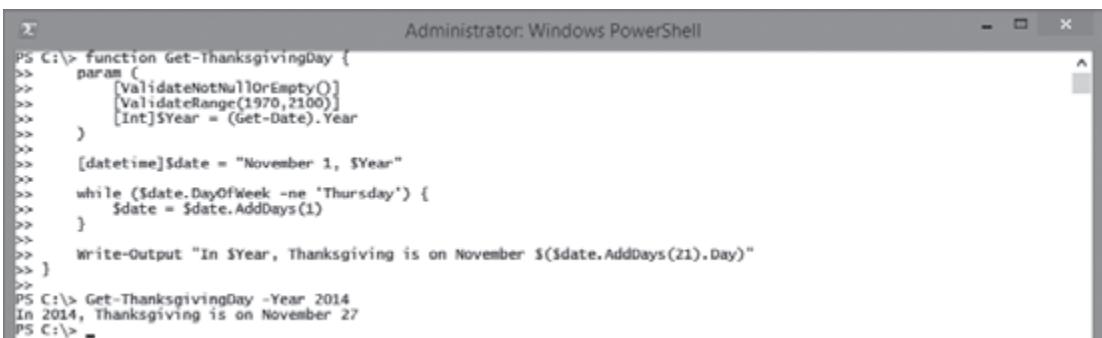
The **While** statement is similar to the **For** statement. This logical construct also executes a block of code as long as some condition is **True**. The syntax of the **While** statement is not as complicated as the **For** statement, as shown in the following example where we've written a real-world function to calculate what day Thanksgiving Day falls on in the United States.

```
function Get-ThanksgivingDay {
    param (
        [ValidateNotNullOrEmpty()]
        [ValidateRange(1970, 2100)]
        [Int]$Year = (Get-Date).Year
    )

    [datetime]$date = "November 1, $Year"

    while ($date.DayOfWeek -ne 'Thursday') {
        $date = $date.AddDays(1)
    }

    Write-Output "In $Year, Thanksgiving is on November $($date.AddDays(21).Day)"
}
```



```
Administrator: Windows PowerShell
PS C:\> function Get-ThanksgivingDay {
>>     param (
>>         [ValidateNotNullOrEmpty()]
>>         [ValidateRange(1970,2100)]
>>         [Int]$Year = (Get-Date).Year
>>     )
>>
>>     [datetime]$date = "November 1, $Year"
>>
>>     while ($date.DayOfWeek -ne 'Thursday') {
>>         $date = $date.AddDays(1)
>>     }
>>
>>     Write-Output "In $Year, Thanksgiving is on November $($date.AddDays(21).Day)"
>> }
>>
PS C:\> Get-ThanksgivingDay -Year 2014
In 2014, Thanksgiving is on November 27
PS C:\> =
```

Figure 13-16

We use a **While** loop in Figure 13-16 to loop through the days of the month of November until we find the first Thursday, and then we know that Thanksgiving is three weeks from that day, since it is always on the fourth Thursday of November.

## Using the Do/While loop

A variation of **While** is **Do While**. In the **While** operation, PowerShell checks the condition at the beginning of the statement. In the **Do While** operation, PowerShell tests the condition at the end of the loop.

```
$i=0
do {
    $i++
    Write-Output "`$i=$i"
}
while ($i -le 5)
```

Figure 13-17

In Figure 13-17, you can see what happens when you check the condition at the end of the statement. The loop essentially says “Increase \$i by 1 and display the current value as long as \$i is less than or equal to 5.”

Notice that we end up with a sixth pass. This occurs because when \$i=5, the **While** condition is still **True**, so the loop repeats, including running the increment and displaying the code. This type of loop will always run at least once until the **While** clause is evaluated. It will continue to loop for as long as the condition is **True**.

## Using the Do/Until loop

A similar loop is **Do Until**. Like **Do While**, PowerShell evaluates the expression at the end of the loop. We thought we would use another real-world example where it's desirable to evaluate the condition at the end of the statement, instead of at the beginning. This example also combines many of the different topics that we've already covered in this chapter.

This PowerShell function is a number-guessing game and as you can see, there is a lot happening inside of the **Do Until** loop.

```

function Start-NumberGame {
    [int]$guess = 0
    [int]$attempt = 0
    [int]$number = Get-Random -Minimum 1 -Maximum 100
    do {
        try {
            $guess = Read-Host "What's your guess?"
            if ($guess -lt 1 -or $guess -gt 100) {
                throw
            }
        }
        catch {
            Write-Output "Invalid number. Numbers 1 through 100 are valid."
            continue
        }
        if ($guess -lt $number) {
            Write-Output "Too low!"
        }
        elseif ($guess -gt $number) {
            Write-Output "Too high!"
        }
        $attempt += 1
    }
    until ($guess -eq $number -or $attempt -eq 7)

    if ($guess -eq $number) {
        Write-Output "You guessed my secret number!"
    }
    else {
        Write-Output "Your out of guesses! Better luck next time!
                    The secret number was $number"
    }
}

```

```

Administrator: Windows PowerShell
PS C:\> function Start-NumberGame {
>>>     [int]$guess = 0
>>>     [int]$attempt = 0
>>>     [int]$number = Get-Random -Minimum 1 -Maximum 100
>>>     do {
>>>         try {
>>>             $guess = Read-Host "What's your guess?"
>>>             if ($guess -lt 1 -or $guess -gt 100) {
>>>                 throw
>>>             }
>>>         }
>>>         catch {
>>>             Write-Output "Invalid number. Numbers 1 through 100 are valid."
>>>             continue
>>>         }
>>>         if ($guess -lt $number) {
>>>             Write-Output "Too low!"
>>>         }
>>>         elseif ($guess -gt $number) {
>>>             Write-Output "Too high!"
>>>         }
>>>         $attempt += 1
>>>     }
>>>     until ($guess -eq $number -or $attempt -eq 7)

>>>     if ($guess -eq $number) {
>>>         Write-Output "You guessed my secret number!"
>>>     }
>>>     else {
>>>         Write-Output "Your out of guesses! Better luck next time!
>>>                     The secret number was $number"
>>>     }
>>> }
>>>
PS C:\> Start-NumberGame
What's your guess?: 50
Too low!
What's your guess?: 75
Too low!
What's your guess?: 87
Too high!
What's your guess?: 81
Too low!
What's your guess?: 84
Too high!
What's your guess?: 83
Too high!
What's your guess?: 82
You guessed my secret number!
PS C:\>

```

Figure 13-18

**Do Until** is a little different as it will keep looping as long as the condition is **False**, that is until the expression is **True**, otherwise a **Do Until** loop is almost the same as a **Do While** loop. The advantage of using **Do Until** is that the loop ends when we expect it to instead of running one additional time. There's nothing wrong with using **Do** loops instead of **While**, but it all depends on what you're trying to accomplish.

In case you hadn't noticed, we also introduced you to the **Continue** keyword in the previous code example, which we'll be covering, along with the **Break** keyword, in more detail in the next section.

## Break and Continue

The **Break** statement terminates just about any logic construct that we've covered in this chapter, including **For**, **Foreach**, **While**, and **Switch**. When a **Break** statement is encountered, PowerShell exits the loop and runs the next command in the block or script, as shown in Figure 13-19.

```
$i = 0
$var = 10,20,30,40

foreach ($val in $var)
{
    $i++
    if ($val -eq 30){
        break
    }
}
Write-Output "Found a match at item $i"
```

```
Administrator: Windows PowerShell
PS C:\> $i=0
PS C:\> $var=10,20,30,40
PS C:\>
PS C:\> foreach ($val in $var)
>> {
>>     $i++
>>     if ($val -eq 30){
>>         break
>>     }
>> }
>> Write-Output "Found a match at item $i"
>>
Found a match at item 3
PS C:\> =
```

Figure 13-19

You can refer back to the **Switch** section of this chapter to see another example of the **Break** statement.

The **Continue** statement is essentially the opposite of **Break**. When the **Continue** statement is encountered, PowerShell returns immediately to the beginning of a loop like **For**, **Foreach**, and **While**. You can also use **Continue** with the **Switch** statement.

```
$var = "PowerShell123", "PowerShell", "123", "PowerShell 123"

Switch -regex ($var) {
    "\w" { Write-Output $_ matches \w" ;
    continue }
    "\d" { Write-Output $_ matches \d" ;
    continue }
    "\s" { Write-Output $_ matches \s" ;
    continue }
    Default { Write-Output "No match found for$_. ;
}
}
```

```
Administrator: Windows PowerShell
PS C:\> $var='PowerShell123','PowerShell','123','PowerShell 123'
PS C:\>
PS C:\> Switch -regex ($var) {
>>     "\w" {Write-Output $_ matches \w" ;
>>     continue}
>>     "\d" {Write-Output $_ matches \d" ;
>>     continue}
>>     "\s" {Write-Output $_ matches \s" ;
>>     continue}
>>     Default {Write-Output "No match found for$_. ;
>>     }
>> }
>>
PowerShell123 matches \w
PowerShell matches \w
123 matches \w
PowerShell 123 matches \w
PS C:\> -
```

Figure 13-20

Refer back to the **Do Until** section of this chapter for another example of using the **Continue** statement.

## Exercise 13 – Making your script smarter

### Task 1

Store your age in a variable and design an **If** statement that displays “You’re too young” if the age is less than 18, “You’re in the groove” if the age is 18 through 64, and “Enjoy your retirement” if 65 and older.

### Task 2

Repeat Task 1 using the **Switch** statement instead of the **If** statement.

## Chapter 14

# Increasing management with WMI/CIM

### What is WMI?

Windows Management Instrumentation (WMI) is Microsoft's implementation of Web-Based Enterprise Management (WBEM).

WMI is a separate technology from PowerShell, but PowerShell provides an interface, in the form of cmdlets, to access WMI. WMI is important because from within PowerShell, you can work with WMI to retrieve and manipulate information related to various aspects of Windows and the hardware that Windows runs on. PowerShell doesn't replace WMI—it uses it.

### WMI structure

WMI is built around classes, which are abstract descriptions of computer-related things. For example, the **Win32\_Volume** class describes what a logical disk volume looks like. The class includes properties like size and serial number, and can perform tasks like defragmenting. However, the class doesn't represent an actual volume—it's just a description of what a volume might look like. When you actually have a volume, you have an instance of the class. For example, if your computer has two volumes, you have two instances of the **Win32\_Volume** class. Each instance has properties such as size and name. It might also have methods such as **Defragment()** that you can use to manage that instance.

There are many WMI classes. Windows itself has hundreds, and products like SQL Server and Microsoft Exchange Server can each add hundreds more. To help keep things structured, classes are organized into what are called namespaces. The main namespace for the core Windows classes is `root\cimv2`.

## Get-WMIOObject versus Get-CIMInstance

**Get-WMIOObject** is one of several older PowerShell cmdlets used for working with WMI and that has been around since PowerShell version 1. A list of these older WMI cmdlets is shown in Figure 14-1.

```
Get-Command -Noun Wmi*
```

```
Administrator: Windows PowerShell
PS C:\> Get-Command -Noun Wmi*
 CommandType      Name
 ----           ----
 Cmdlet          Get-WmiObject
 Cmdlet          Invoke-WmiMethod
 Cmdlet          Register-WmiEvent
 Cmdlet          Remove-WmiObject
 Cmdlet          Set-WmiInstance
 ModuleName
 -----
 Microsoft.PowerShell.Management
 Microsoft.PowerShell.Management
 Microsoft.PowerShell.Management
 Microsoft.PowerShell.Management
 Microsoft.PowerShell.Management
 PS C:\>
```

Figure 14-1

**Get-WMIOObject** uses DCOM to communicate with WMI on remote machines, and while there have been minor updates to this cmdlet in both PowerShell versions 3 and 4, we consider it to be more or less deprecated. Whenever possible, we try to use the **Get-CimInstance** cmdlet that was introduced in PowerShell version 3.

**Get-CimInstance** defaults to communicating with WSMAN instead of DCOM, when working with remote computers, but it can also use DCOM when used in combination with a CIM session. One huge benefit of using WSMAN instead of DCOM is that it's more firewall friendly, but in order to use the CIM cmdlets with WSMAN, the remote computer has to be running at least version 3 of the WSMAN stack. You can determine the stack version of a remote computer that has PowerShell remoting enabled by running the **Test-WSMan** cmdlet against it.

```
Test-WSMan -ComputerName dc01
```

```
Administrator: Windows PowerShell
PS C:\> Test-WSMan -ComputerName dc01
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
PS C:\>
```

Figure 14-2

A list of the CIM cmdlets that are available beginning with PowerShell version 3 are shown in Figure 14-3.

```
Get-Command -Module CimCmdlets
```

CommandType	Name	ModuleName
Cmdlet	Export-BinaryMiLog	CimCmdlets
Cmdlet	Get-CimAssociatedInstance	CimCmdlets
Cmdlet	Get-CimClass	CimCmdlets
Cmdlet	Get-CimInstance	CimCmdlets
Cmdlet	Get-CimSession	CimCmdlets
Cmdlet	Import-BinaryMiLog	CimCmdlets
Cmdlet	Invoke-CimMethod	CimCmdlets
Cmdlet	New-CimInstance	CimCmdlets
Cmdlet	New-CimSession	CimCmdlets
Cmdlet	New-CimSessionOption	CimCmdlets
Cmdlet	Register-CimIndicationEvent	CimCmdlets
Cmdlet	Remove-CimInstance	CimCmdlets
Cmdlet	Remove-CimSession	CimCmdlets
Cmdlet	Set-CimInstance	CimCmdlets

Figure 14-3

One of our favorite features of **Get-CimInstance** over **Get-WMIOBJECT** is tabbed expansion of namespaces and classes.

## Navigating and using WMI/CIM

When it comes to PowerShell, we always recommend using the built-in Help instead of searching the Internet. However, when it comes to WMI, there's very little documentation in PowerShell, since it's a separate technology. If you're searching the Internet for how to accomplish a task in WMI and happen to come across a VBScript example, the query itself is a simple copy and paste directly from VBScript to PowerShell, which we will demonstrate.

The following is a VBScript example.

```

Dim strComputer
Dim strOutput
Dim objWMIS
Dim colWMI
Dim objProcessor

strComputer = "."

On Error Resume Next

Set objWMIS = GetObject("winmgmts:\\" & strComputer & "\root\cimv2")
Set colWMI = objWMIS.ExecQuery("SELECT DeviceID, MaxClockSpeed, Name, Status FROM Win32_Processor")

For Each objProcessor In colWMI
    strOutput = strOutput & objProcessor.Name & ":" & objProcessor.DeviceID & "-Speed: " _
        & objProcessor.MaxClockSpeed & "Mhz (Status:" & objProcessor.Status & ")" & vbCrLf
Next

wscript.echo strOutput

```

## Windows PowerShell: TFM

We can simply take the query out of that VBScript example and use it with the **-Query** parameter of **Get-WMIObject** or **Get-CimInstance**.

```
Get-CimInstance -Query "SELECT DeviceID, MaxClockSpeed, Name, Status FROM Win32_Processor"
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command run is "Get-CimInstance -Query 'SELECT DeviceID, MaxClockSpeed, Name, Status FROM Win32\_Processor'". The output is a table:

DeviceID	Name	Caption	MaxClockSpeed	SocketDesignation	Manufacturer
CPU0	Intel(R) Core...		2592		

Figure 14-4

## Navigating with the WMI Explorer

Consider downloading a tool like SAPIEN WMI Explorer ([www.sapien.com](http://www.sapien.com)), which allows you to see all the namespaces on your computer and browse their classes and instances, as shown in Figure 14-5.

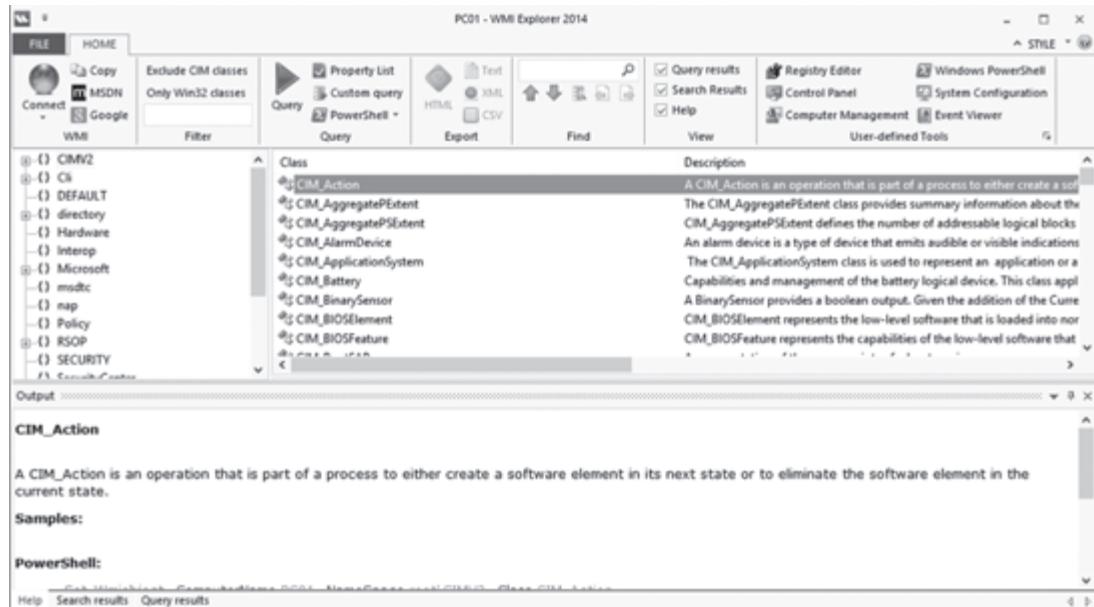


Figure 14-5

## Working with classes

The discoverability of WMI classes is built right into PowerShell. If you're using the older WMI cmdlets, **Get-WMIOObject** has a **-List** parameter that retrieves the WMI classes for a particular namespace, with `root\cimv2` being the default if no namespace is specified. If you use this parameter and don't filter down the results, be prepared to be overwhelmed.

```
Get-WMIOObject -List
```

Name	Methods	Properties
CIM_Indication	{}	{CorrelatedIndications, IndicationFilterName, IndicationId...}
CIM_ClassIndication	{}	ClassDefinition, CorrelatedIndications, IndicationFilterNa...
CIM_ClassDeletion	{}	ClassDefinition, CorrelatedIndications, IndicationFilterNa...
CIM_ClassCreation	{}	ClassDefinition, CorrelatedIndications, IndicationFilterNa...
CIM_ClassModification	{}	ClassDefinition, CorrelatedIndications, IndicationFilterNa...
CIM_InstIndication	{}	CorrelatedIndications, IndicationFilterName, IndicationIde...
CIM_InstCreation	{}	CorrelatedIndications, IndicationFilterName, IndicationIde...
CIM_InstModification	{}	CorrelatedIndications, IndicationFilterName, IndicationIde...
CIM_InstDeletion	{}	CorrelatedIndications, IndicationFilterName, IndicationIde...
_NotifyStatus	{}	CorrelatedIndications, IndicationFilterName, IndicationIde...
_ExtendedStatus	{}	CorrelatedIndications, IndicationFilterName, IndicationIde...
Win32_PrivilegesStatus	{}	Description, Operation, ParameterInfo, ProviderName...}
Win32_JobObjectStatus	{}	Description, Operation, ParameterInfo, PrivilegesNotHeld...}
CIM_Error	{}	AdditionalDescription, Description, Operation, ParameterIn...
MSFT_WmiError	{}	CIMStatusCode, CIMStatusCodeDescription, ErrorSource, Err...
MSFT_ExtendedStatus	{}	CIMStatusCode, CIMStatusCodeDescription, error_Category, e...
_SecurityRelatedClass	{}	CIMStatusCode, CIMStatusCodeDescription, error_Category, e...
_Trustee	{}	Domain, Name, SID, SidLength...}
Win32_Trustee	{}	Domain, Name, SID, SidLength...}
_NTLMUser9X	{}	Authority, Flags, Mask, Name...}
_ACE	{}	AccessMask, AceFlags, AceType, GuidInheritedObjectType...}
Win32_ACE	{}	AccessMask, AceFlags, AceType, GuidInheritedObjectType...}
_SecurityDescriptor	{}	ControlFlags, DACL, Group, Owner...}
Win32_SecurityDescriptor	{}	ControlFlags, DACL, Group, Owner...}
PARAMETERS	{}	
_SystemClass	{}	
_ProviderRegistration	{}	provider]
_EventProviderRegistration	{}	EventQueryList, provider]
_ObjectProviderRegistration	{}	InteractionType, provider, QuerySupportLevels, SupportsB...
_ClassProviderRegistration	{}	CacheRefreshInterval, InteractionType, PerUserSchema, prov...
_InstanceProviderRegistration	{}	InteractionType, provider, QuerySupportLevels, SupportsB...
_MethodProviderRegistration	{}	provider]
_PropertyProviderRegistration	{}	provider, SupportsGet, SupportsPut]
_EventConsumerProviderRegistration	{}	ConsumerClassName, provider]
thisNAMESPACE	{}	SECURITY_DESCRIPTOR]
NAMESPACE	{}	Name)
_IndicationRelated	{}	)
_FilterToConsumerBinding	{}	Consumer, CreatorSID, DeliverSynchronously, DeliveryQoS...)
_EventConsumer	{}	(CreatorSID, MachineName, MaximumQueueSize)

Figure 14-6

We've found that most of the time the classes you're going to want begin with **Win32\_\***, and unless you're looking for performance information, you won't need the ones that begin with **Win32\_Perf\***.

```
Get-WmioObject -List -Class Win32_* | Where-Object name -notlike Win32_perf*
```

## Windows PowerShell: TFM

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command run is "PS C:\> Get-WmiObject -List -Class Win32\_\* | Where-Object name -notlike Win32\_perf\*". The output is a table with three columns: "Name", "Methods", and "Properties". The "Name" column lists various WMI class names. The "Methods" column shows methods like Reboot, Shutdown, Create, Terminate, SetPowerState, etc., each associated with a parameter set (e.g., R...). The "Properties" column lists properties for each class, such as EventType, SECURITY\_DESCRIPTOR, TIME\_CREATED, ParentProcessID, ProcessID, ProcessName, SECURITY\_DESCRIPTOR, etc.

Name	Methods	Properties
Win32_DeviceChangeEvent		{EventType, SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_SystemConfigurationChangeEvent		{EventType, SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_VolumeChangeEvent		{DriveName, EventType, SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_SystemTrace		{SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_ProcessTrace		
Win32_ProcessStartTrace		ParentProcessID, ProcessID, ProcessName, SECURITY_DESCRIPTOR...
Win32_ProcessStopTrace		ParentProcessID, ProcessID, ProcessName, SECURITY_DESCRIPTOR...
Win32_ThreadTrace		ExitStatus, ParentProcessID, ProcessID, ProcessName...}
Win32_ThreadStartTrace		processID, SECURITY_DESCRIPTOR, ThreadID, TIME_CREATED}
Win32_ThreadStopTrace		processID, SECURITY_DESCRIPTOR, StackBase, StackLimit...}
Win32_ModuleTrace		processID, SECURITY_DESCRIPTOR, ThreadID, TIME_CREATED}
Win32_ModuleLoadTrace		SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_PowerManagementEvent		DefaultBase, FileName, ImageBase, ImageChecksum...}
Win32_ComputerSystemEvent		EventType, OEMEventCode, SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_ComputerShutdownEvent		MachineName, SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_IPRouteTableEvent		MachineName, SECURITY_DESCRIPTOR, TIME_CREATED, Type}
Win32_OperatingSystem		SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_Process	Reboot, Shutdown...	BootDevice, BuildNumber, BuildType, Caption...}
Win32_ComputerSystem	Create, Terminat...	Caption, Commandline, CreationClassName, CreationDate...}
Win32_NTDomain		AdminPasswordStatus, AutomaticManagedPagefile, AutomaticRe...
Win32_BIOS		Caption, ClientSiteName, CreationClassName, DCSiteName...}
Win32_SoftwareElement		BiosCharacteristics, BIOSVersion, BuildNumber, Caption...}
Win32_VideoController		Attributes, BuildNumber, Caption, CodeSet...}
Win32_SCSIController	SetPowerState, R...	AcceleratorCapabilities, AdapterCompatibility, AdapterDACT...
Win32_InfraredDevice	SetPowerState, R...	Availability, Caption, ConfigManagerErrorCode, ConfigManag...
Win32_PCMCIAController	SetPowerState, R...	Availability, Caption, ConfigManagerErrorCode, ConfigManag...
Win32_FloppyController	SetPowerState, R...	Availability, Caption, ConfigManagerErrorCode, ConfigManag...
Win32_USBController	SetPowerState, R...	Availability, Caption, ConfigManagerErrorCode, ConfigManag...
Win32_SerialPort	SetPowerState, R...	Availability, Caption, ConfigManagerErrorCode, ConfigManag...
Win32_ParallelPort	SetPowerState, R...	Availability, Binary, Capabilities, CapabilityDescriptions...
Win32_IDEController	SetPowerState, R...	Availability, Capabilities, CapabilityDescriptions, Captio...
Win32_1394Controller	SetPowerState, R...	Availability, Caption, ConfigManagerErrorCode, ConfigManag...
Win32_TemperatureProbe	SetPowerState, R...	Availability, Caption, ConfigManagerErrorCode, ConfigManag...
Win32_VoltageProbe	SetPowerState, R...	Accuracy, Availability, Caption, ConfigManagerErrorCode...
Win32_CurrentProbe	SetPowerState, R...	Accuracy, Availability, Caption, ConfigManagerErrorCode...
Win32_Bus	SetPowerState, R...	SetPowerState, R... Accuracy, Availability, Caption, ConfigManagerErrorCode...
Win32_Keyboard	SetPowerState, R...	SetPowerState, R... Availability, BusNum, BusType, Caption...}
Win32/DesktopMonitor	SetPowerState, R...	SetPowerState, R... Availability, Caption, ConfigManagerErrorCode, ConfigManag...
Win32_PointingDevice	SetPowerState, R...	SetPowerState, R... Availability, Bandwidth, Caption, ConfigManagerErrorCode...}
		SetPowerState, R... Availability, Caption, ConfigManagerErrorCode, ConfigManag...

Figure 14-7

The CIM cmdlets have a dedicated cmdlet for discovering the WMI classes that you're looking for. The following example returns the same results as the command that we ran in Figure 14-7, using the **Get-WmiObject** cmdlet.

```
Get-CimClass -ClassName Win32_* | Where-Object CimClassName -notlike Win32_perf*
```

## Increasing management with WMI/CIM

Administrator: Windows PowerShell		
NameSpace: ROOT/CIMV2	CimClassMethods	CimClassProperties
Win32_DeviceChangeEvent	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, EventType}
Win32_SystemConfigurationChangeEvent	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, EventType}
Win32_VolumeChangeEvent	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, EventType, DriveName}
Win32_SystemTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_ProcessTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ParentProcessID, ProcessID}
Win32_ProcessStartTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ParentProcessID, ProcessID}
Win32_ProcessStopTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ProcessID, ThreadID}
Win32_ThreadTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ProcessID, ThreadID...}
Win32_ThreadStartTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ProcessID, ThreadID}
Win32_ThreadStopTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ProcessID, ThreadID}
Win32_ModuleTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_ModuleLoadTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, DefaultBase, FileName...}
Win32_PowerManagementEvent	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, EventType, OEMEventCode}
Win32_ComputerSystemEvent	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, MachineName}
Win32_ComputerShutdownEvent	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, MachineName, Type}
Win32_IP4RouteTableEvent	{}	{SECURITY_DESCRIPTOR, TIME_CREATED}
Win32_OperatingSystem	Reboot, Shutdown...	{Caption, Description, InstallDate, Name...}
Win32_Process	Create, Terminat...	{Caption, Description, InstallDate, Name...}
Win32_ComputerSystem	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_NTDomain	{}	{Caption, Description, InstallDate, Name...}
Win32_BIOS	{}	{Caption, Description, InstallDate, Name...}
Win32_SoftwareElement	{}	{Caption, Description, InstallDate, Name...}
Win32_VideoController	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_SCSIController	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_InfraredDevice	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_PCMCIAController	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_FloppyController	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_USBController	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_SerialPort	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_ParallelPort	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_IDEController	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_1394Controller	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_TemperatureProbe	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_VoltageProbe	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_CurrentProbe	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_Bus	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_Keyboard	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32/DesktopMonitor	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_PointingDevice	SetPowerState, R...	{Caption, Description, InstallDate, Name...}

Figure 14-8

The examples we've shown so far don't really narrow the results down to anything specific. If you're looking for information that has to do with, say, a serial number, use a modified version of the command that we used in Figure 14-8, and specify the **-PropertyName** parameter with **SerialNumber** as the value for that parameter.

```
Get-CimClass -ClassName Win32_* -PropertyName SerialNumber
```

Administrator: Windows PowerShell		
NameSpace: ROOT/cimv2	CimClassMethods	CimClassProperties
Win32_OperatingSystem	{Reboot, Shutdown...}	{Caption, Description, InstallDate, Name...}
Win32_BIOS	{}	{Caption, Description, InstallDate, Name...}
Win32_SoftwareElement	{}	{Caption, Description, InstallDate, Name...}
Win32_DiskDrive	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_CDROMDrive	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_Volume	SetPowerState, R...	{Caption, Description, InstallDate, Name...}
Win32_PhysicalMedia	{}	{Caption, Description, InstallDate, Name...}
Win32_PhysicalMemory	{}	{Caption, Description, InstallDate, Name...}
Win32_OnBoardDevice	{}	{Caption, Description, InstallDate, Name...}
Win32_SystemEnclosure	{IsCompatible}	{Caption, Description, InstallDate, Name...}
Win32_BaseBoard	{IsCompatible}	{Caption, Description, InstallDate, Name...}
Win32_PhysicalMemoryArray	{IsCompatible}	{Caption, Description, InstallDate, Name...}
Win32_SystemSlot	{}	{Caption, Description, InstallDate, Name...}
Win32_PortConnector	{}	{Caption, Description, InstallDate, Name...}

Figure 14-9

There are probably several items in this list that you already knew had a serial number and probably a few that you didn't know about.

**Get-CimClass** also has a **-MethodName** parameter that can be used to discover or search for cmdlets that have specific methods.

```
Get-CimClass -ClassName Win32_* -MethodName GetOwner
```

```
Administrator: Windows PowerShell
PS C:\> Get-CimClass -ClassName Win32_* -MethodName GetOwner
NameSpace: ROOT/cimv2
CimClassName          CimClassMethods      CimClassProperties
-----          -----      -----
Win32_Process          {Create, Terminate...} {Caption, Description, InstallDate, Name...}

PS C:\>
```

Figure 14-10

We can see in the previous example that the **Win32\_Process** class is the only class with a method named **GetOwner**. Let's see all the methods that are available for the **Win32\_Process** class.

```
(Get-CimClass -ClassName Win32_Process | Select-Object -ExpandProperty CimClassMethods).name
```

```
Administrator: Windows PowerShell
PS C:\> (Get-CimClass -ClassName Win32_Process | Select-Object -ExpandProperty CimClassMethods).name
Create
Terminate
GetOwner
GetOwnerId
SetPriority
AttachDebugger
GetAvailableVirtualSize
PS C:\>
```

Figure 14-11

## Working with WMI/CIM objects and properties

We're now going to use the **Get-CIMInstance** cmdlet to retrieve information about services. You may be wondering why you wouldn't just use the **Get-Service** cmdlet? There's usually additional information available through WMI that's not available through the standard cmdlets, such as **Get-Service**. We're going to retrieve the **ProcessId** for all of the services that are currently running.

```
Get-CimInstance -ClassName Win32_Service -Filter "State = 'Running'" |
Select-Object DisplayName, Name, ProcessId
```

## Increasing management with WMI/CIM

Administrator: Windows PowerShell

```
PS C:\> Get-CimInstance -ClassName Win32_Service -Filter "State = 'Running'" |
>> Select-Object DisplayName, Name, ProcessId
```

DisplayName	Name	ProcessId
Windows Audio Endpoint Builder	AudioEndpointBuilder	452
Windows Audio	Audiosrv	900
Base Filtering Engine	BFE	1168
Background Tasks Infrastructure Service	BrokerInfrastructure	748
Certificate Propagation	CertPropSvc	924
Cryptographic Services	CryptSvc	660
DCOM Server Process Launcher	DcomLaunch	748
DHCP Client	Dhcp	900
DNS Client	DnsCache	660
Diagnostic Policy Service	DPS	1168
Device Setup Manager	DsmSvc	924
Windows Event Log	EventLog	900
COM+ Event System	EventSystem	992
Windows Font Cache Service	FontCache	992
IP Helper	iphlpvc	924
Server	LanmanServer	924
Workstation	LanmanWorkstation	660
TCP/IP NetBIOS Helper	lmhosts	900
Local Session Manager	LSM	748
Windows Firewall	MpsSvc	1168
Network Connection Broker	NcbService	452
Netlogon	Netlogon	672
Network List Service	netprofm	992
Network Location Awareness	NlaSvc	660
Network Store Interface Service	nsi	992
Program Compatibility Assistant Service	PcaSvc	452
Plug and Play	PlugPlay	748
Power	Power	748
User Profile Service	ProfSvc	924
RPC Endpoint Mapper	RpcEptMapper	788
Remote Procedure Call (RPC)	RpcSs	788
Security Accounts Manager	SamSs	672
Smart Card Device Enumeration Service	ScDeviceEnum	452
Task Scheduler	Schedule	924
System Event Notification Service	SENS	924
Remote Desktop Configuration	SessionEnv	924
Shell Hardware Detection	ShellHdetection	924
Print Spooler	Spooler	1144
Superfetch	SysMain	452
System Events Broker	SystemEventsBroker	748

Figure 14-12

Notice that we used the **-Filter** parameter to filter left, instead of retrieving all of the services and then piping them to **Where-Object**.

We could have used the **-Query** parameter, along with WQL (WMI Query Language), to retrieve the same information.

```
Get-CimInstance -Query "Select * from Win32_Service where State = 'Running'" |
Select-Object DisplayName, Name, ProcessId
```

## Windows PowerShell: TFM

Administrator: Windows PowerShell

```
PS C:\> Get-CimInstance -Query "Select * from Win32_Service where State = 'Running'" | Select-Object DisplayName, Name, ProcessId
```

DisplayName	Name	ProcessId
Windows Audio Endpoint Builder	AudioEndpointBuilder	452
Windows Audio	Audiosrv	900
Base Filtering Engine	BFE	1168
Background Tasks Infrastructure Service	BrokerInfrastructure	748
Certificate Propagation	CertPropSvc	924
Cryptographic Services	CryptSvc	660
DCOM Server Process Launcher	DcomLaunch	748
DHCP Client	Dhcp	900
DNS Client	DnsCache	660
Diagnostic Policy Service	DPS	1168
Device Setup Manager	DsmSvc	924
Windows Event Log	EventLog	900
COM+ Event System	EventSystem	992
Windows Font Cache Service	FontCache	992
IP Helper	iphlpvc	924
Server	LanmanServer	924
Workstation	LanmanWorkstation	660
TCP/IP NetBIOS Helper	lmhosts	900
Local Session Manager	LSM	748
Windows Firewall	MpsSvc	1168
Network Connection Broker	NcbService	452
Netlogon	Netlogon	672
Network List Service	netprofm	992
Network Location Awareness	NlaSvc	660
Network Store Interface Service	nsi	992
Program Compatibility Assistant Service	PcaSvc	452
Plug and Play	PlugPlay	748
Power	Power	748
User Profile Service	ProfSvc	924
RPC Endpoint Mapper	RpcEptMapper	788
Remote Procedure Call (RPC)	RpcSs	788
Security Accounts Manager	SamSs	672
Smart Card Device Enumeration Service	ScDeviceEnum	452
Task Scheduler	Schedule	924
System Event Notification Service	SENS	924
Remote Desktop Configuration	SessionEnv	924
Shell Hardware Detection	ShellHdDetection	924
Print Spooler	Spooler	1144
Superfetch	SysMain	452
System Events Broker	SystemEventsBroker	748

Figure 14-13

## Executing WMI/CIM methods

We now want to backup and clear our application event log, so let's see what classes are available for working with event logs.

```
Get-CimClass -ClassName Win32_*EventLog*
```

Administrator: Windows PowerShell

```
PS C:\> Get-CimClass -ClassName Win32_EventLog*
```

NameSpace: ROOT/cimv2	CimClassName	CimClassMethods	CimClassProperties
	Win32_NTEventLogFile	{TakeOwnership, C...	{Caption, Description, InstallDate, Name...}
	Win32_NTLogEventLog	{}	{Log, Record}

Figure 14-14

We'll use the same process that we previously used, to determine what methods are available for the **Win32\_NTEventLogFile** class.

## Increasing management with WMI/CIM

```
Get-CimClass -ClassName Win32_NTEventLogFile | Select-Object -ExpandProperty CimClassMethods
```

Name	ReturnType	Parameters	Qualifiers
---	-----	-----	-----
TakeOwnership	UInt32 []	{Option, SecurityDescriptor}	{Schema, ValueMap}
ChangeSecurityPermissions	UInt32 []	{FileName}	{Schema, ValueMap}
Copy	UInt32 []	{FileName}	{Schema, ValueMap}
Rename	UInt32 []	{}	{Schema, ValueMap}
Delete	UInt32 []	{}	{Schema, ValueMap}
Compress	UInt32 []	{}	{Schema, ValueMap}
Uncompress	UInt32 []	{}	{Schema, ValueMap}
TakeOwnershipEx	UInt32 []	{Recursive, StartFileName, ...}	{Schema, ValueMap}
ChangeSecurityPermissionsEx	UInt32 []	{Option, Recursive, Securi...}	{Schema, ValueMap}
CopyEx	UInt32 []	{FileName, Recursive, Star...}	{Schema, ValueMap}
DeleteEx	UInt32 []	{StartFileName, StopFileName}	{Schema, ValueMap}
CompressEx	UInt32 []	{Recursive, StartFileName, ...}	{Schema, ValueMap}
UncompressEx	UInt32 []	{Recursive, StartFileName, ...}	{Schema, ValueMap}
GetEffectivePermission	Boolean	{Permissions}	[Privileges, Schema]
ClearEventlog	UInt32	{ArchiveFileName}	[implemented, Privileges, ...]
BackupEventlog	UInt32	{ArchiveFileName}	[implemented, Privileges, ...]

Figure 14-15

We see that there are **BackupEventLog** and **ClearEventLog** methods—and those are exactly what we're looking for. We'll use the **Win32\_NTEventLogFile** class and the **BackupEventLog** method to back up the application event log.

```
Get-CimInstance -ClassName Win32_NTEventLogFile -Filter "LogFileName = 'Application'" |
Invoke-CimMethod -Name BackupEventLog -Arguments @{
ArchiveFileName = "c:\tmp\application_backup.evtx" }
```

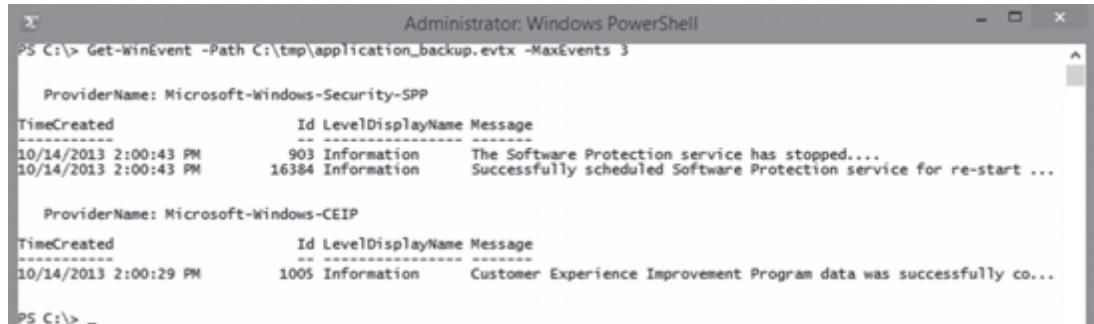
ReturnValue	PSComputerName
0	

Figure 14-16

Then we'll verify that we can query the backup of our application event log.

```
Get-WinEvent -Path C:\tmp\application_backup.evtx -MaxEvents 3
```

## Windows PowerShell: TFM



Administrator: Windows PowerShell

```
PS C:\> Get-WinEvent -Path C:\tmp\application_backup.evtx -MaxEvents 3

ProviderName: Microsoft-Windows-Security-SPP
TimeCreated          Id LevelDisplayName Message
-----          --  -----      -----
10/14/2013 2:00:43 PM    903 Information   The Software Protection service has stopped...
10/14/2013 2:00:43 PM   16384 Information  Successfully scheduled Software Protection service for re-start ...

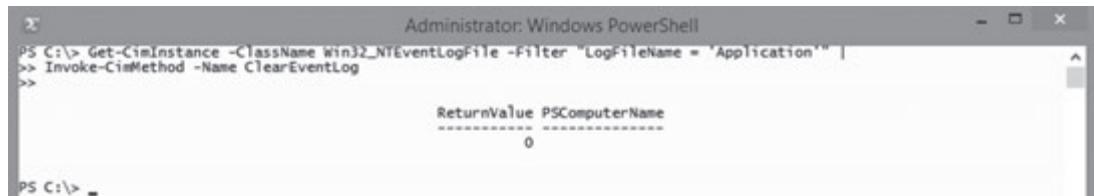
ProviderName: Microsoft-Windows-CEIP
TimeCreated          Id LevelDisplayName Message
-----          --  -----      -----
10/14/2013 2:00:29 PM   1005 Information  Customer Experience Improvement Program data was successfully co...

PS C:\> _
```

Figure 14-17

We'll now use the **ClearEventLog** method to clear the application event log, since we've already backed it up and verified that the backup contains data.

```
Get-CimInstance -ClassName Win32_NTEventLogFile -Filter "LogFileName = 'Application'" |
Invoke-CimMethod -Name ClearEventLog
```



Administrator: Windows PowerShell

```
PS C:\> Get-CimInstance -ClassName Win32_NTEventLogFile -Filter "LogFileName = 'Application'" |
>> Invoke-CimMethod -Name ClearEventLog
>>

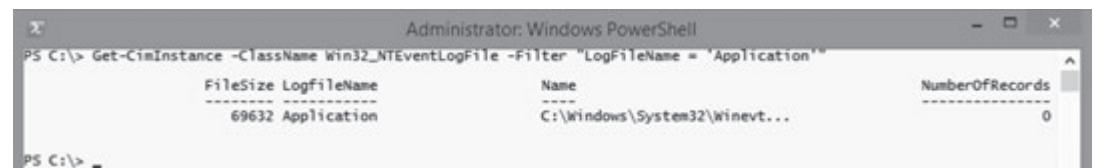
ReturnValue PSComputerName
----- -----
0

PS C:\> _
```

Figure 14-18

We'll now query the **Win32\_NTEventLogFile** WMI class and filter our results down to only the application event log to determine how many records it contains.

```
Get-CimInstance -ClassName Win32_NTEventLogFile -Filter "LogFileName = 'Application'"
```



Administrator: Windows PowerShell

```
PS C:\> Get-CimInstance -ClassName Win32_NTEventLogFile -Filter "LogFileName = 'Application'" | Select-Object FileSize, LogfileName, Name, NumberOfRecords
Filesize LogfileName      Name      NumberOfRecords
-----  -----      -----
69632 Application  C:\Windows\System32\winevt... 0

PS C:\> _
```

Figure 14-19

As you can see in Figure 14-19, there are zero records in the application event log. So, the log was indeed cleared.

## Increasing your reach to remote computers

Many of the CIM cmdlets have a **-ComputerName** parameter for working with remote computers.

We covered the **Get-CimClass** cmdlet earlier in this chapter, but you can also use that cmdlet to discover namespaces and classes on remote machines.

```
Get-CimClass -ComputerName web01 -Namespace root/MicrosoftIISv2 -ClassName IIsApp*
```

```
Administrator: Windows PowerShell
PS C:\> Get-CimClass -ComputerName web01 -Namespace root/MicrosoftIISv2 -ClassName IIsApp*
NameSpace: root/MicrosoftIISv2
CimClassName          CimClassMethods      CimClassProperties
-----          {EnumAppsInPool, ...} {Caption, Description, InstallDate, Name...}
IIsApplicationPool   {}                   {Caption, Description, InstallDate, Name...}
IIsApplicationPools {}                   {Caption, Description, SettingID, Name...}
IIsApplicationPoolSetting {}               {Caption, Description, SettingID, Name...}
IIsApplicationPool_IIsAdminACL {}         {GroupComponent, PartComponent}
IIsApplicationPools_IIsAdminACL {}         {GroupComponent, PartComponent}
IIsApplicationPools_IIsApplicati... {}       {GroupComponent, PartComponent}
IIsApplicationPool_IIsApplicatio... {}       {Element, Setting}
IIsApplicationPools_IIsApplicati... {}       {Element, Setting}
```

Figure 14-20

How many times in the past have you had to make a trip to a remote datacenter, called someone at a remote datacenter, or purchased a third-party product to retrieve information such as the serial number on servers or workstations at your remote locations?

Earlier in this chapter, we also discovered that WMI classes can provide serial number information. We'll now use that information to retrieve the serial number from each of our server VMs.

```
Get-CimInstance -ComputerName dc01, sql01, web01 -ClassName Win32_BIOS |
Select-Object -Property PSComputerName, SerialNumber
```

```
Administrator: Windows PowerShell
PS C:\> Get-CimInstance -ComputerName dc01, sql01, web01 -ClassName Win32_BIOS |
>> Select-Object -Property PSComputerName, SerialNumber
>>
PSComputerName          SerialNumber
-----          -----
web01                  0793-5816-4052-9088-9374-7310-89
dc01                  0274-6338-1852-6443-8657-2526-32
sql01                  1654-6460-9122-4570-2105-9647-49
```

Figure 14-21

You may be thinking that those look like strange serial numbers, but that's because those servers are virtual machines running on Hyper-V. If they were physical servers, it would be the actual serial number or service tag of the server.

## Security and end-points with WMI

One major difference between **Get-WMIOObject** and **Get-CimInstance** is that **Get-WMIOObject** has a **-Credential** parameter for specifying alternate credentials, and **Get-CimInstance** does not.

We recommend running PowerShell as a domain user, and at most as a local admin, which means the session of PowerShell you're running won't have access to authenticate and to run PowerShell commands against remote servers. So how do you overcome this security obstacle?

**Get-CimInstance**, along with many other native cmdlets and functions that were added with PowerShell version 3, can use a **CimSession**, which is created with the **New-CimSession** cmdlet. It's also more efficient to create a **CimSession** if you will be running multiple commands against the same servers requesting information over and over again. This is because the overhead for creating a **CimSession** is only incurred once, instead of each time when specifying a remote computer via the **-ComputerName** parameter.

We're now going to demonstrate creating a **CimSession** to our three virtual servers, using the default credentials that we're running PowerShell as.

```
$CimSession = New-CimSession -ComputerName dc01, sql01, web01
```



Figure 14-22

Notice that although the **CimSessions** we created are stored in a single variable, there are three separate **CimSessions** and not one **CimSession** to three servers.

```
$CimSession
```

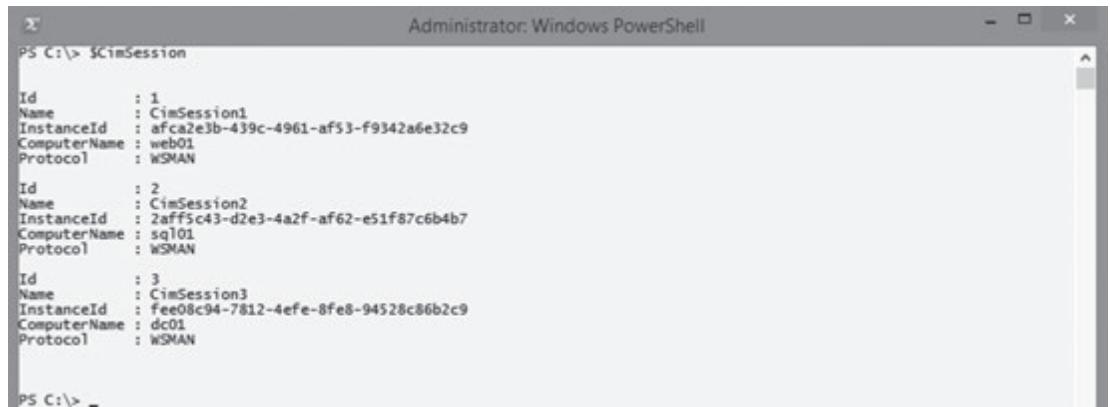
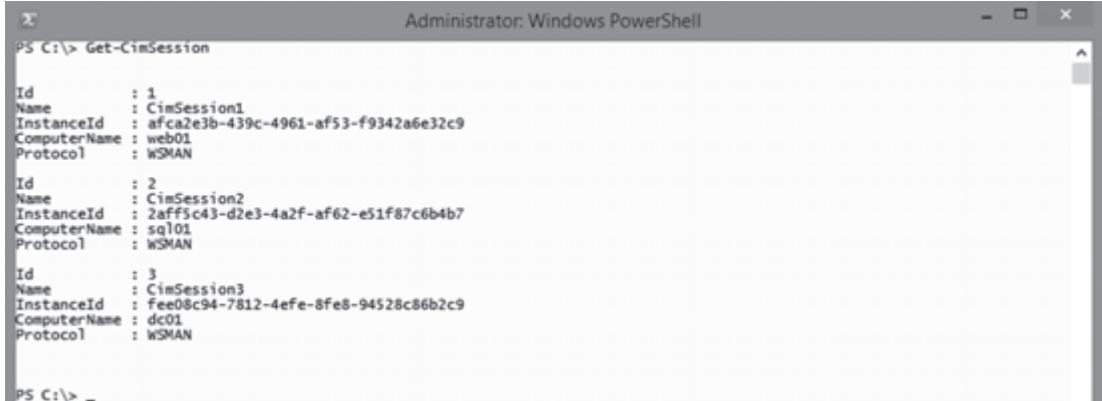


Figure 14-23

We can use the **Get-CimSession** cmdlet to show all of the current CimSessions.

```
Get-CimSession
```



```
Administrator: Windows PowerShell
PS C:\> Get-CimSession

Id      : 1
Name    : CimSession1
InstanceId : afca2e3b-439c-4961-af53-f9342a6e32c9
ComputerName : web01
Protocol   : WSMAN

Id      : 2
Name    : CimSession2
InstanceId : 2afffc43-d2e3-4a2f-af62-e51f87c6b4b7
ComputerName : sql01
Protocol   : WSMAN

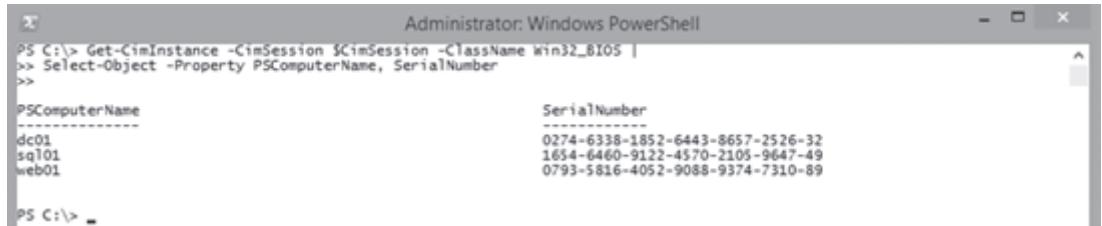
Id      : 3
Name    : CimSession3
InstanceId : fee08c94-7812-4efe-8fe8-94528c86b2c9
ComputerName : dc01
Protocol   : WSMAN

PS C:\> _
```

Figure 14-24

We'll now run the a similar command to what we previously ran to retrieve the serial number information from our three servers, but this time we'll use the CimSessions that were created, instead of their computer names.

```
Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS |
Select-Object -Property PSComputerName, SerialNumber
```



```
Administrator: Windows PowerShell
PS C:\> Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS |
>> Select-Object -Property PSComputerName, SerialNumber
>>

PSComputerName          SerialNumber
-----          -----
dc01                    0274-6338-1852-6443-8657-2526-32
sql01                  1654-6460-9122-4570-2105-9647-49
web01                  0793-5816-4052-9088-9374-7310-89

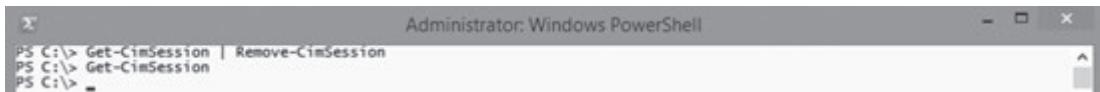
PS C:\> _
```

Figure 14-25

We'll remove all of the current CimSessions, and then confirm that we have no CimSessions connected.

```
Get-CimSession | Remove-CimSession
Get-CimSession
```

### Windows PowerShell: TFM

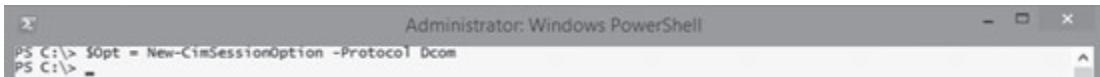


```
Administrator: Windows PowerShell
PS C:\> Get-CimSession | Remove-CimSession
PS C:\> Get-CimSession
PS C:\> =
```

Figure 14-26

We'll use the **New-CimSessionOption** cmdlet to create an option that can use the DCOM protocol for connecting to older clients that either don't have PowerShell installed or aren't running version 3 of the WSMAN stack.

```
$0pt = New-CimSessionOption -Protocol Dcom
```



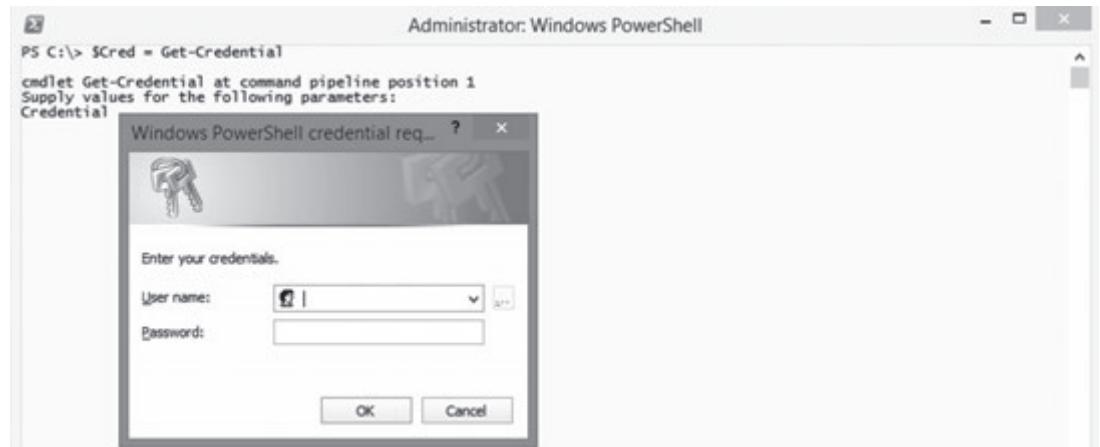
```
Administrator: Windows PowerShell
PS C:\> $0pt = New-CimSessionOption -Protocol Dcom
PS C:\> =
```

Figure 14-27

It's worth noting that a CimSession is required to use the DCOM protocol with **Get-CimInstance**.

We'll now store our alternate credentials in a variable.

```
$Cred = Get-Credential
```

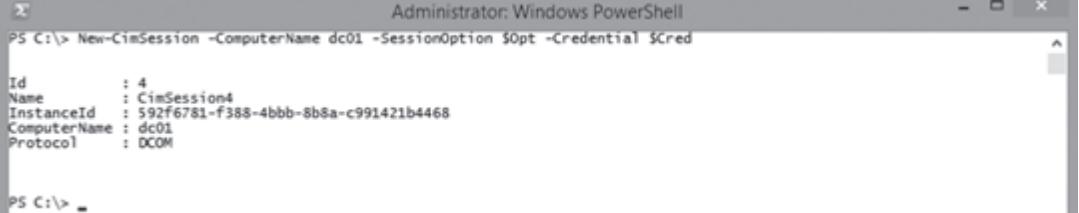


```
Administrator: Windows PowerShell
PS C:\> $Cred = Get-Credential
cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
Windows PowerShell credential req... ? ×
Enter your credentials.
User name: [ ] OK Cancel
Password: [ ]
```

Figure 14-28

We'll create a CimSession to our dc01 server using the DCOM protocol and alternate credentials.

```
New-CimSession -ComputerName dc01 -SessionOption $Opt -Credential $Cred
```



```
Administrator: Windows PowerShell
PS C:\> New-CimSession -ComputerName dc01 -SessionOption $Opt -Credential $Cred
Id      : 4
Name    : CimSession4
InstanceId : 592f6781-f388-4bbb-8b8a-c991421b4468
ComputerName : dc01
Protocol : DCOM

PS C:\> _
```

Figure 14-29

To show you the flexibility of CimSessions, we'll create a CimSession to our sql01 server, by using alternate credentials, but using the default WSMAN protocol.

```
New-CimSession -ComputerName sql01 -Credential $Cred
```



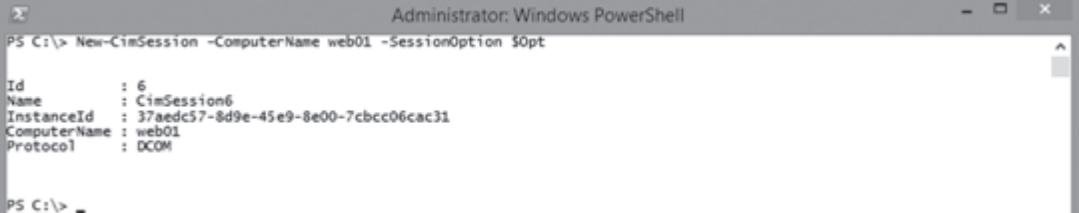
```
Administrator: Windows PowerShell
PS C:\> New-CimSession -ComputerName sql01 -Credential $Cred
Id      : 5
Name    : CimSession5
InstanceId : c9c8850e-ebc5-4063-b712-013fae18aea3
ComputerName : sql01
Protocol : WSMAN

PS C:\> _
```

Figure 14-30

We'll also create a CimSession to our web01 server using the DCOM protocol, but use the default credentials that we're running PowerShell as.

```
New-CimSession -ComputerName web01 -SessionOption $Opt
```



```
Administrator: Windows PowerShell
PS C:\> New-CimSession -ComputerName web01 -SessionOption $Opt
Id      : 6
Name    : CimSession6
InstanceId : 37aedc57-8d9e-45e9-8e00-7cbcc06cac31
ComputerName : web01
Protocol : DCOM

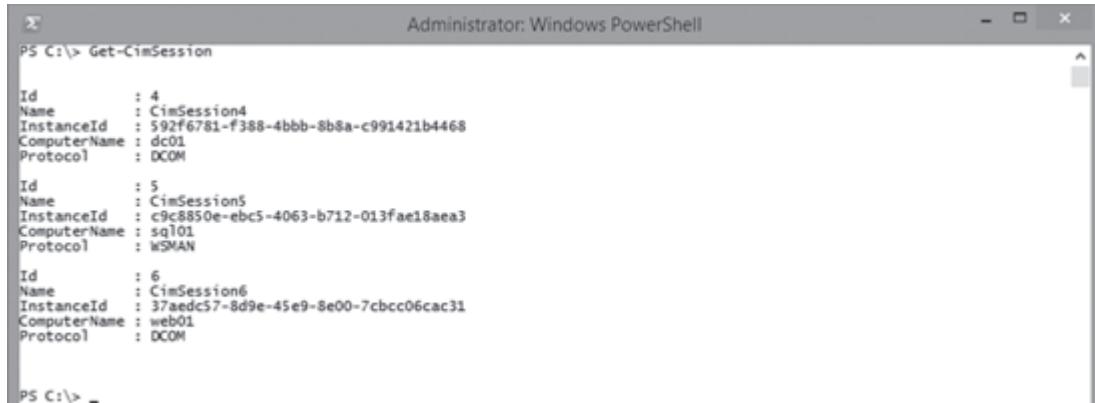
PS C:\> _
```

Figure 14-31

### Windows PowerShell: TFM

You can see that we now have three CimSessions established, one to each one of our remote virtual servers.

```
Get-CimSession
```



```
Administrator: Windows PowerShell
PS C:\> Get-CimSession

Id      : 4
Name    : CimSession4
InstanceId : 592f6781-f388-4bbb-8b8a-c991421b4468
ComputerName : dc01
Protocol   : DCOM

Id      : 5
Name    : CimSession5
InstanceId : c9c8850e-ebc5-4063-b712-013fae18aea3
ComputerName : sq101
Protocol   : WSMAN

Id      : 6
Name    : CimSession6
InstanceId : 37aedc57-8d9e-45e9-8e00-7cbcc06cac31
ComputerName : web01
Protocol   : DCOM

PS C:\> =
```

Figure 14-32

We'll store those three CimSessions in a variable and confirm that the variable does indeed contain them.

```
$CimSession = Get-CimSession
$CimSession
```



```
Administrator: Windows PowerShell
PS C:\> $CimSession = Get-CimSession
PS C:\> $CimSession

Id      : 4
Name    : CimSession4
InstanceId : 592f6781-f388-4bbb-8b8a-c991421b4468
ComputerName : dc01
Protocol   : DCOM

Id      : 5
Name    : CimSession5
InstanceId : c9c8850e-ebc5-4063-b712-013fae18aea3
ComputerName : sq101
Protocol   : WSMAN

Id      : 6
Name    : CimSession6
InstanceId : 37aedc57-8d9e-45e9-8e00-7cbcc06cac31
ComputerName : web01
Protocol   : DCOM

PS C:\> =
```

Figure 14-33

We'll now run the same command as before and retrieve the serial number information again.

```
Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS |
Select-Object -Property PSComputerName, SerialNumber
```

```
Administrator: Windows PowerShell
PS C:\> Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS |
>> Select-Object -Property PSComputerName, SerialNumber
>>
PSComputerName          SerialNumber
-----          -----
dc01                    0274-6338-1852-6443-8657-2526-32
web01                  0793-5816-4052-9088-9374-7310-89
sql01                  1654-6460-9122-4570-2105-9647-49
PS C:\> _
```

Figure 14-34

Think about what's going on in Figure 14-34. The command that we ran uses a different CimSession to communicate with each server, as it always does, but each CimSession has different settings. Two use DCOM, two use alternate credentials, one uses native credentials, and one uses the default WSMAN protocol.

We think it's really neat how this works and it's all transparent for the most part, unless you really think about what's going on.

We briefly mentioned that there are other cmdlets and functions that were introduced beginning with PowerShell version 3 that can take advantage of CimSessions as well. **Get-Volume** is one of those functions and we'll use it to query the fixed disk information on our three servers.

```
Get-Volume -CimSession $CimSession | Where-Object DriveType -eq Fixed
```

```
Administrator: Windows PowerShell
PS C:\> Get-Volume -CimSession $CimSession | Where-Object DriveType -eq Fixed
DriveLetter FileSystemLabel FileSystem DriveType HealthStatus SizeRemaining Size PSComputerName
-----          -----          -----          -----
S              System Rese... NTFS      Fixed     Healthy    90.24 MB   350 MB dc01
S              Share             NTFS      Fixed     Healthy    120.05 GB   127 GB dc01
C              System Rese... NTFS      Fixed     Healthy    121.37 GB   126.66 GB dc01
C              System Rese... NTFS      Fixed     Healthy    90.24 MB   350 MB sql01
C              System Rese... NTFS      Fixed     Healthy    90.24 MB   350 MB web01
C              NTFS               Fixed     Healthy    122.13 GB   126.66 GB web01
C              NTFS               Fixed     Healthy    120.07 GB   126.66 GB sql01
PS C:\> _
```

Figure 14-35

We want you to understand how important learning the concept of CimSessions are, so we're going to show you just how many cmdlets on our Windows 8.1 workstation have CimSession parameters. Keep in mind that we do have the Remote Server Administration Tools (RSAT) installed, which accounts for many of these results.

We start out by importing all of our modules, because checking which cmdlets have with a particular parameter doesn't trigger the module auto-loading feature.

```
Get-Module -ListAvailable | Import-Module -DisableNameChecking
```

Figure 14-36

Now we'll simply get a count of the cmdlets with a **CimSession** parameter, by using the **Get-Command** cmdlet.

```
(Get-Command -ParameterName CimSession).Count
```

Figure 14-37

We have over a thousand cmdlets with a **CimSession** parameter. We'll also retrieve the total number of cmdlets on our machine just to put that number into perspective.

```
(Get-Command).Count
```

Figure 14-38

We have over 2200 cmdlets on our workstation, and as you can see, almost half of them have a **CimSession** parameter. What does that mean? It means that CimSessions should definitely be on the list of PowerShell things you want to learn and learn well.

## Exercise 14 – Increasing management with WMI

### Task 1

Without using the **Get-Volume** cmdlet that we demonstrated in this chapter, retrieve a list of fixed disks in the computers in your test lab environment using WMI. Include the total size and remaining free space of the drive.

### Task 2

Determine the number of logical processors and the amount of system memory in the computers in your test lab environment using WMI.



## Chapter 15

# Super-sized management with PowerShell remoting

### Why you need this now

Administering servers and workstations by logging into each one locally or via Remote Desktop, and manually clicking through the GUI, is not a very efficient process and it's not something that's reliably repeatable. You can't replicate pointing and clicking to update ten production systems at two o'clock in the morning, based on pointing and clicking in the previous day's test environment. A well-crafted PowerShell script, though, can be run in a test environment, and then scheduled to run at night to update those ten (or even 100) production systems, and only wake you up if there's a problem.

With the default installation type being Server Core (no GUI) on Windows Server 2012 and higher, logging in locally or via remote desktop won't buy you anything except a black command prompt window. You can accomplish everything remotely with PowerShell that you can accomplish with that command prompt, without having the overhead of a remote desktop session.

Figure out how to accomplish a task in PowerShell for one server and you can use the same process to accomplish the same task on hundreds or thousands of servers just as easily—by taking advantage of the power of PowerShell remoting.

### The Windows Management Framework

PowerShell is pre-installed on all modern Windows operating systems, but depending on what operating system version you have, you may want to update to a newer version of PowerShell for additional functionality. Newer versions of PowerShell are distributed as part of the Windows Management Framework. PowerShell version 4 is distributed as part of the Windows Management Framework 4.0.

Before installing the Windows Management Framework on any machine, make sure to check the system requirements and check for possible application compatibility issues. For example, as of this writing, the Windows Management Framework 3.0 isn't yet supported on Windows Servers that are running Exchange Server versions prior to Microsoft Exchange Server 2013, per the Exchange Team Blog: <http://tinyurl.com/anchobz>.

## How remoting works—the WS-MAN protocol

Windows Remote Management (WinRM) is the Microsoft implementation of the Web Services for Management (WS-MAN) protocol, also known as WS-Management, which is used by Windows PowerShell to allow commands to be invoked on remote computers.

By default, you must be a member of the administrators group on the remote computer to connect to it through PowerShell remoting.

## Enabling PowerShell remoting

The **Enable-PSRemoting** cmdlet is used to configure a computer to receive remote commands.

On server operating systems beginning with Windows Server 2012, PowerShell remoting is enabled by default, so enabling it is unnecessary on those operating systems.

To enable PowerShell remoting, first open PowerShell as an administrator on the destination machine that you want to be able to run remote commands against, and note that the title bar says **Administrator: Windows PowerShell**, as shown in Figure 15-1. It's important to run PowerShell as an administrator since it is unable to participate in user access control.



Figure 15-1

Run the **Enable-PSRemoting** cmdlet and select A (for all) at both of the prompts.

```
Enable-PSRemoting
```

```

Administrator: Windows PowerShell
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"): a
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 the operation "Set-PSSessionConfiguration" on target "Name: microsoft.powershell SDQL: O:NG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)". This lets selected users 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"): a
PS C:\>

```

Figure 15-2

As shown in Figure 15-2, there are a number of items that are configured by running the **Enable-PSRemoting** cmdlet.

If you want to enable PowerShell remoting without receiving these confirmation prompts, you can enable it like a PowerShell Jedi, by using the **-Force** parameter.

```
Enable-PSRemoting -Force
```

```

Administrator: Windows PowerShell
PS C:\> Enable-PSRemoting -Force
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.
PS C:\>

```

Figure 15-3

#### Note:

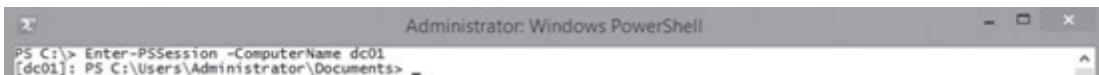
Although it is possible to run PowerShell version 2 on a machine with PowerShell version 3 or version 4 installed, make sure to enable PowerShell remoting from the highest number version of PowerShell that's installed on the machine.

It is also possible to enable PowerShell remoting through group policy, but that topic is outside the scope of what will be covered in this chapter.

## Managing one-to-one

The **Enter-PSSession** cmdlet is used to create an interactive session with an individual remote computer. One-to-one remoting reminds us of how telnet works, because you connect to a remote computer, as shown in Figure 15-4, and once connected, the prompt is prefixed with the name of the remote computer that you're connected to.

```
Enter-PSSession -ComputerName dc01
```

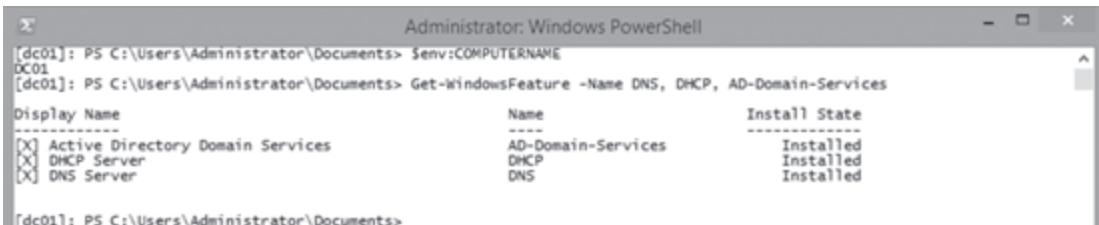


```
Administrator: Windows PowerShell
PS C:\> Enter-PSSession -ComputerName dc01
[dc01]: PS C:\Users\Administrator\Documents>
```

Figure 15-4

Any commands that are run while you're in a one-to-one remoting session are actually run on the remote computer, just as if you were logged in locally on that computer and running the commands.

```
$env:COMPUTERNAME
Get-WindowsFeature -Name DNS, DHCP, AD-Domain-Services
```



Display Name	Name	Install State
[X] Active Directory Domain Services	AD-Domain-Services	Installed
[X] DHCP Server	DHCP	Installed
[X] DNS Server	DNS	Installed

```
Administrator: Windows PowerShell
[dc01]: PS C:\Users\Administrator\Documents> $env:COMPUTERNAME
DC01
[dc01]: PS C:\Users\Administrator\Documents> Get-WindowsFeature -Name DNS, DHCP, AD-Domain-Services
Display Name           Name           Install State
-----             ----           -----
[X] Active Directory Domain Services   AD-Domain-Services   Installed
[X] DHCP Server          DHCP          Installed
[X] DNS Server            DNS           Installed
[dc01]: PS C:\Users\Administrator\Documents>
```

Figure 15-5

To end a one-to-one remoting session, run the **Exit-PSSession** cmdlet.

```
Exit-PSSession
```



```
Administrator: Windows PowerShell
[dc01]: PS C:\Users\Administrator\Documents> Exit-PSSession
PS C:\>
```

Figure 15-6

## Managing one-to-many

The **Invoke-Command** cmdlet is used to perform one-to-many remoting, also known as fan-out remoting, and it's one of the primary cmdlets that you'll use to "super size" the management of your remote machines with PowerShell. While this cmdlet is designed for one-to-many remoting, it can also be used with a single remote computer, although not interactively like with the **Enter-PSSession** cmdlet. **Invoke-Command** can also be run against the local computer as long as PowerShell remoting is enabled on it.

In its simplest form, the remote computers are specified via the **-ComputerName** parameter and the command to run on the remote computer is enclosed in the **ScriptBlock**.

```
Invoke-Command -ComputerName dc01, sql01, web01 -ScriptBlock { $env:COMPUTERNAME }
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "Invoke-Command -ComputerName dc01, sql01, web01 -ScriptBlock { \$env:COMPUTERNAME }". The output displayed is "DC01", "WEB01", and "SQL01", each on a new line. The PowerShell prompt "PS C:\>" is visible at the bottom.

Figure 15-7

In Figure 15-7, the credentials that PowerShell is running as will be used to authenticate to the remote computers. If you would like to use alternate credentials for authenticating to the remote computers, use the **-Credential** parameter to specify them.

```
Invoke-Command -ComputerName dc01, sql01, web01 { Get-Culture } -Credential (Get-Credential)
```

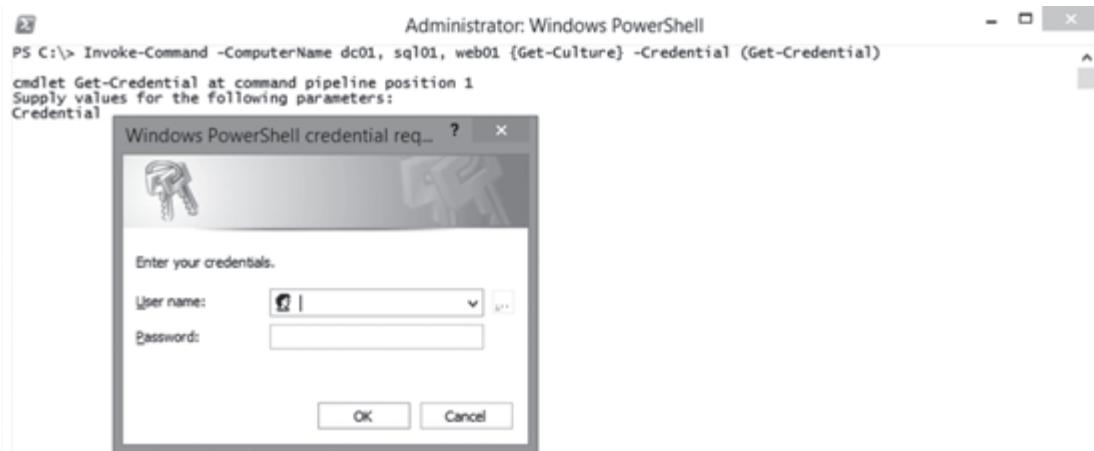


Figure 15-8

It's a common occurrence to see the actual name of the **-ScriptBlock** parameter omitted, since it's a positional parameter.

One point of confusion is that the **ScriptBlock** doesn't have to be on one line. If you need to run multiple, separate commands inside of the script block, such as importing a module before executing a command, there's no reason to put it all on one line with semicolons separating it. Break it up into multiple lines, design your code for readability, and you'll thank us later because your code will be much easier to read and troubleshoot.

```
Invoke-Command -ComputerName dc01 -ScriptBlock {
    Import-Module -Name ActiveDirectory
    Get-ADUser -Identity Administrator
}
```

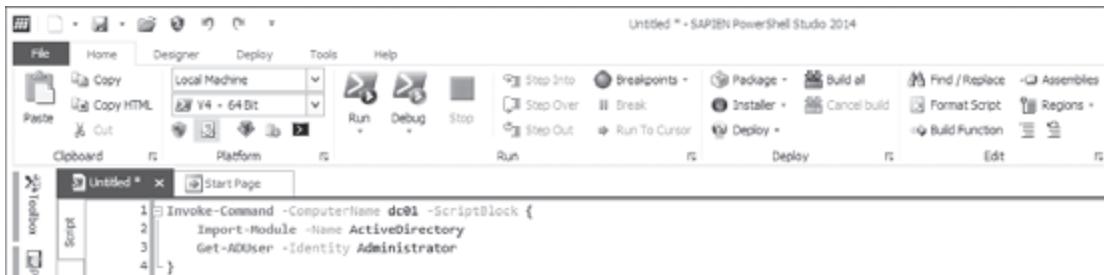


Figure 15-9

This is possible even if you're working interactively in the PowerShell console.

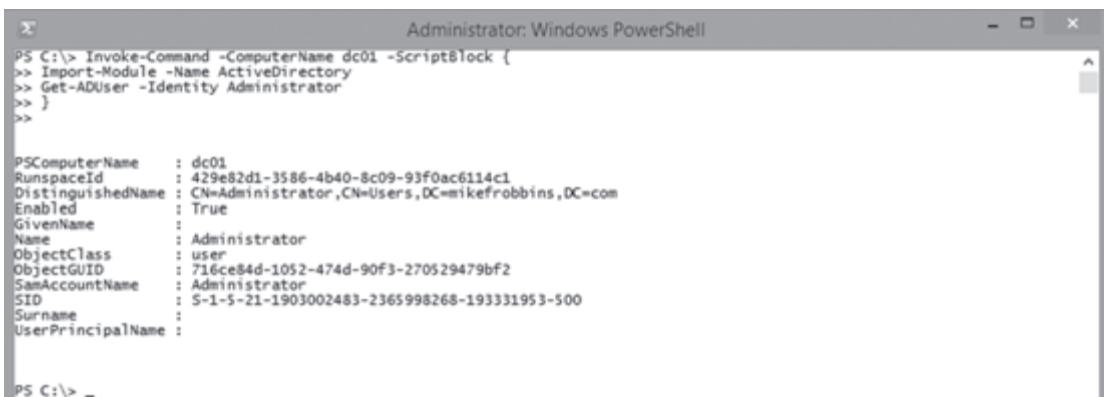


Figure 15-10

As shown in Figure 15-10, when using the **Invoke-Command** cmdlet, an additional synthetic property named **PSCoMPuterName** is added to the output, which helps you to keep track of which com-

puter the results are from. There's a **-HideComputerName** parameter if you prefer not to display this property.

By default, the **Invoke-Command** cmdlet will run against 32 remote computers at a time, in parallel. This value can be increased or decreased on a per command basis, by using the **-ThrottleLimit** parameter, although this doesn't change its default value—it only changes it for that particular command.

```
Invoke-Command -ComputerName dc01, sql01, web01 { Get-Culture } -ThrottleLimit 2
```

LCID	Name	DisplayName	PSComputerName
1033	en-US	English (United States)	dc01
1033	en-US	English (United States)	sql01
1033	en-US	English (United States)	web01

Figure 15-11

Notice that, in Figure 15-11, we throttled the number of computers that we ran the previous command against to two at a time, instead of the default of 32.

## Remote command output

The output of most of the remote commands that we've demonstrated, with the exception of the ones from one-to-one remoting, are deserialized XML objects.

```
Invoke-Command -ComputerName dc01 { Get-Process } | Get-Member
```

```
Administrator: Windows PowerShell
PS C:\> Invoke-Command -ComputerName dc01 {Get-Process} | Get-Member
TypeName: Deserialized.System.Diagnostics.Process
```

Figure 15-12

This means they are inert objects that don't contain methods, so you can't use their methods to perform an action on them, such as stop a service by using the **Stop** method. With services, you could just use the **Stop-Service** cmdlet, but there may be a method you want to use that doesn't have a corresponding cmdlet.

## Establishing sessions

You can also pre-create a session with the **New-PSSession** cmdlet, and then use that session with either the **Enter-PSSession** or **Invoke-Command** cmdlet.

Since the **Enter-PSSession** cmdlet is used for one-to-one remoting, we'll pre-create a session to a single computer, and then use that session to establish an interactive session to the computer named **dc01**.

```
$session = New-PSSession -ComputerName dc01
Enter-PSSession -Session $session
```

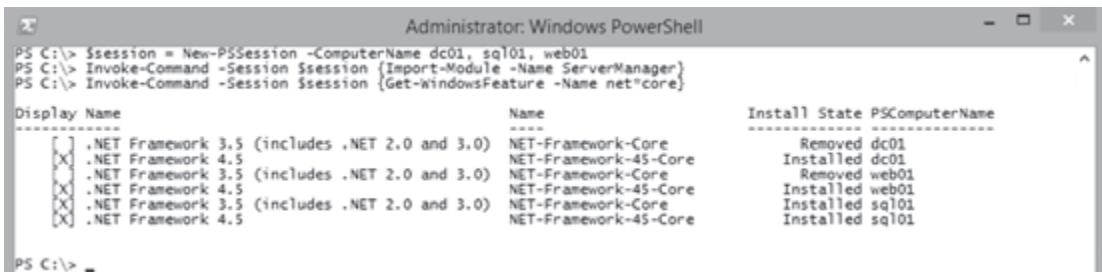


```
Administrator: Windows PowerShell
PS C:\> $session = New-PSSession -ComputerName dc01
PS C:\> Enter-PSSession -Session $session
[dc01]: PS C:\Users\Administrator\Documents> _
```

Figure 15-13

In Figure 15-13, we showed how to pre-create a session to a remote computer. While this didn't buy us much with one-to-one remoting, pre-creating a session allows us to create a persistent connection that we can run command after command across when used with one-to-many remoting, instead of having the overhead of setting up and tearing down a connection for each command that we want to execute against the remote computers.

```
$session = New-PSSession -ComputerName dc01, sql01, web01
Invoke-Command -Session $session { Import-Module -Name ServerManager }
Invoke-Command -Session $session { Get-WindowsFeature -Name net*core }
```



```
Administrator: Windows PowerShell
PS C:\> $session = New-PSSession -ComputerName dc01, sql01, web01
PS C:\> Invoke-Command -Session $session {Import-Module -Name ServerManager}
PS C:\> Invoke-Command -Session $session {Get-WindowsFeature -Name net*core}

Display Name                                Name          Install State PSComputerName
-----                                     ----          -----          -----
[ ] .NET Framework 3.5 (includes .NET 2.0 and 3.0) NET-Framework-Core      Removed   dc01
[X] .NET Framework 4.5                      NET-Framework-45-Core    Installed  dc01
[ ] .NET Framework 3.5 (includes .NET 2.0 and 3.0) NET-Framework-Core      Removed   web01
[X] .NET Framework 4.5                      NET-Framework-45-Core    Installed  web01
[ ] .NET Framework 3.5 (includes .NET 2.0 and 3.0) NET-Framework-Core      Installed  sql01
[X] .NET Framework 4.5                      NET-Framework-45-Core    Installed  sql01

PS C:\> _
```

Figure 15-14

When using PowerShell remoting, design your commands so that as much of the filtering takes place on the remote computer as possible, to minimize the amount of data that's transmitted back across the network to the computer requesting the information. Notice that in Figure 15-14, we filtered left by using the **-Name** parameter of **Get-WindowsFeature** and only brought back two features from each of the remote servers.

Now, we'll show you how NOT to accomplish the same task. While this command looks similar, the

way it accomplishes the task is very different.

```
Invoke-Command -Session $session { Get-WindowsFeature } | Where-Object Name -like net*core
```

Administrator: Windows PowerShell			
Display Name	Name	Install State	PSComputerName
.NET Framework 3.5 (includes .NET 2.0 and 3.0)	NET-Framework-Core	Removed	dc01
.NET Framework 4.5	NET-Framework-45-Core	Installed	dc01
.NET Framework 3.5 (includes .NET 2.0 and 3.0)	NET-Framework-Core	Removed	web01
.NET Framework 4.5	NET-Framework-45-Core	Installed	web01
.NET Framework 3.5 (includes .NET 2.0 and 3.0)	NET-Framework-Core	Installed	sql01
.NET Framework 4.5	NET-Framework-45-Core	Installed	sql01

Figure 15-15

In Figure 15-15, all of the features from each of the servers are transmitted across the network, brought back to the requesting computer, and then filtered down to two items per server. This may not seem like a big deal—until your network admin pays you a visit because you just transmitted over 800 items across the network, which is 13,250 percent more than what was necessary.

```
(Invoke-Command -Session $session { Get-WindowsFeature }).count
```

Administrator: Windows PowerShell	
PS C:\>	(Invoke-Command -Session \$session {Get-WindowsFeature}).count
801	

Figure 15-16

The **Get-PSSession** cmdlet can be used to see your open sessions, their current state, and their availability.

```
Get-PSSession
```

Administrator: Windows PowerShell					
Id	Name	ComputerName	State	ConfigurationName	Availability
10	Session10	sql01	Opened	Microsoft.PowerShell	Available
9	Session9	dc01	Opened	Microsoft.PowerShell	Available
11	Session11	web01	Opened	Microsoft.PowerShell	Available

Figure 15-17

It's a best practice to clean up after yourself, even in PowerShell. Once you're finished with those persistent sessions that you created, they should be cleaned up, otherwise they'll remain open for the life of your PowerShell console or until their idle timeout is reached (the default is two hours).

Use the **Remove-PSSession** cmdlet to remove sessions.

```
Get-PSSession | Remove-PSSession
Get-PSSession
```



```
Administrator: Windows PowerShell
PS C:\> Get-PSSession | Remove-PSSession
PS C:\> Get-PSSession
PS C:\> =
```

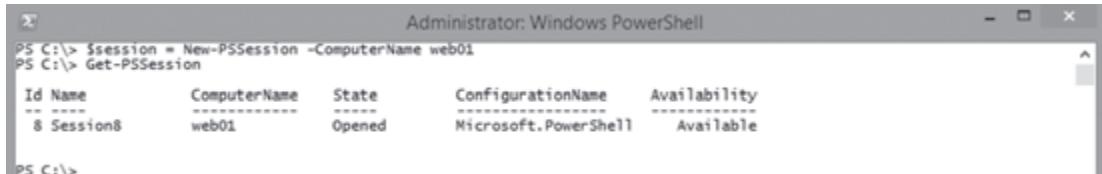
Figure 15-18

We ran the **Get-PSSession** cmdlet after removing our PSSessions, in Figure 15-18, to confirm that the sessions were indeed removed.

## Disconnected sessions

We've created a session to our server named web01, to demonstrate disconnected sessions, which is a new feature introduced in PowerShell version 3.

```
$session = New-PSSession -ComputerName web01
Get-PSSession
```



```
Administrator: Windows PowerShell
PS C:\> $session = New-PSSession -ComputerName web01
PS C:\> Get-PSSession
Id Name           ComputerName   State      ConfigurationName Availability
-- --           -----        -----      -----          -----
  8 Session8       web01         Opened     Microsoft.PowerShell Available
PS C:\> =
```

Figure 15-19

In order to use the disconnected sessions feature in PowerShell, the remote computer must be running at least PowerShell version 3 and if you're going to disconnect a previously connected session, the session must be in an open state. Interactive PSSessions cannot be disconnected.

Disconnected sessions allow you to disconnect from a PSSession and reconnect to it later. The session state is maintained and queued commands will continue to run in the disconnected session. This allows admins to kick off long-running processes, disconnect the PSSession, and then reconnect to it later to check on its status from the same, or even a different, computer, if needed. We'll disconnect from the

Supersized management with PowerShell remoting session we previously created using the **Disconnect-PSSession** cmdlet.

```
Disconnect-PSSession -Session $session
```

Administrator: Windows PowerShell				
Id	Name	ComputerName	State	ConfigurationName
8	Session8	web01	Disconnected	Microsoft.PowerShell
PS C:\>				

Figure 15-20

Use the **Connect-PSSession** cmdlet to reconnect to a disconnected PSSession.

```
Connect-PSSession -Session $session
```

Administrator: Windows PowerShell				
Id	Name	ComputerName	State	ConfigurationName
8	Session8	web01	Opened	Microsoft.PowerShell
PS C:\>				

Figure 15-21

Only the original user who created the disconnected session can reconnect to it. Also, keep in mind that by default a session will only stay disconnected until the duration of its idle timeout value is reached (the default is two hours).

If you want to kick off a long-running process to a server that you don't already have an existing PSSession to and you plan to disconnect the session, you can use the **-InDisconnectedSession** parameter to create a session when starting the command, and it will immediately and automatically disconnect the session once the command is started. We can also use the **-SessionName** parameter to name the session and make it easier to identify when reconnecting to it later.

```
Invoke-Command -ComputerName sql01 { Get-Culture } -InDisconnectedSession -SessionName sqlsession
```

Administrator: Windows PowerShell				
Id	Name	ComputerName	State	ConfigurationName
10	sqlsession	sql01	Disconnected	Microsoft.PowerShell
PS C:\>				

Figure 15-22

Both the **-InDisconnectedSession** and **-SessionName** parameters of the **Invoke-Command** cmdlet are new to PowerShell version 3 and the **-SessionName** parameter is only valid when used in conjunction with the **-InDisconnectedSession** parameter.

The **Receive-PSSession** cmdlet is used to receive the results of the commands that were run while a session was disconnected.

```
Receive-PSSession -Name sqlsession
```

LCID	Name	DisplayName	PSComputerName
1033	en-US	English (United States)	sq101

Figure 15-23

Running the **Receive-PSSession** cmdlet against a disconnected session, as demonstrated in Figure 15-23, also automatically reconnects the disconnected session.

```
Get-PSSession
```

Id	Name	ComputerName	State	ConfigurationName	Availability
10	sqlsession	sq101	Opened	Microsoft.PowerShell	Available
8	Session8	web01	Opened	Microsoft.PowerShell	Available

Figure 15-24

PSSessions must be explicitly disconnected by using one of the methods shown in this portion of the chapter. Otherwise, a PSSession is terminated if PowerShell is closed out or if the computer is shut down.

## Getting cmdlets anytime you need them

We need to query Active Directory from within PowerShell and we don't have the Active Directory PowerShell cmdlets installed on our computer. Maybe we have an operating system that doesn't support the version of the Remote Server Administration Tools that we need to install or maybe we're using a shared computer and it would be a security risk to install them locally.

Implicit Remoting is where a PowerShell module, or specific cmdlets that exist on a remote machine, can be imported locally as functions from a PSSession. These imported commands act and feel just like the real cmdlets, except that they are actually a shortcut to the real cmdlet. When those shortcuts are run, the commands actually run on the remote server instead of locally. Sound interesting?

To use the Active Directory cmdlets via implicit remoting, we'll use the **Import-PSSession** cmdlet to import the Active Directory module from our domain controller named dc01 and create a PSSession on the fly, all in one line. Remember, the portion of the command in parentheses will execute first.

```
Import-PSSession -Session (New-PSSession -ComputerName dc01) -Module ActiveDirectory
```

ModuleType	Version	Name	ExportedCommands
Script	1.0	tmp_wqvuglax.0hz	{Add-ADCentralAccessPolicyMember, Add-ADComputerServiceAcc...

Figure 15-25

We'll use the **Get-Command** cmdlet to show that these commands are indeed functions.

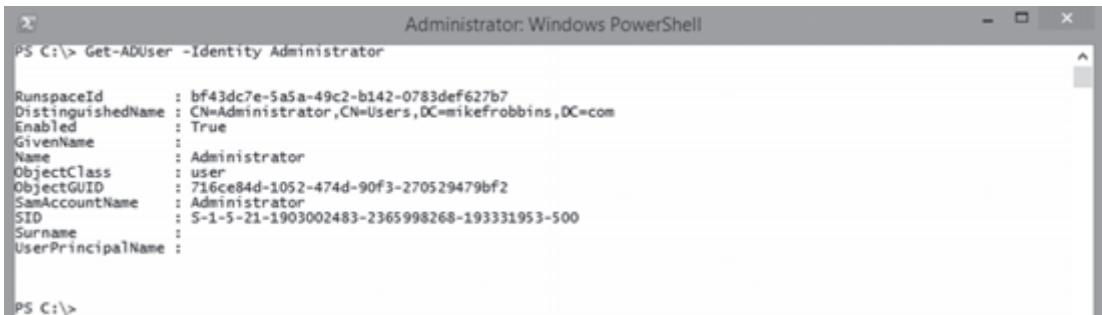
```
Get-Command -Name Get-ADUser
```

CommandType	Name	ModuleName
Function	Get-ADUser	tmp_wqvuglax.0hz

Figure 15-26

We can now use the commands just as if they were installed locally.

```
Get-ADUser -Identity Administrator
```



```
Administrator: Windows PowerShell
PS C:\> Get-ADUser -Identity Administrator

RunspaceId : bf43dc7e-5a5a-49c2-b142-0783def627b7
DistinguishedName : CN=Administrator,CN=Users,DC=mikefrobbins,DC=com
Enabled : True
GivenName :
Name : Administrator
ObjectClass : user
ObjectGUID : 716ce84d-1052-474d-90f3-270529479bf2
SamAccountName : Administrator
SID : S-1-5-21-1903002483-2365998268-193331953-500
Surname :
UserPrincipalName : 
```

Figure 15-27

Remember that you are still working with deserialized objects, so keep in mind that you're not working with live objects. This is an important distinction between these functions and the actual cmdlets.

While we have access to the Active Directory cmdlets, we'll go ahead and create a user named Ferdinand, a group named Technicians, and then add Ferdinand to the Technicians group.

```
New-ADUser -SamAccountName 'ferdinand' -Name 'Ferdinand' -Enabled $true ` 
-AccountPassword (Read-Host "Enter User Password" -AsSecureString)

New-ADGroup -Name 'Technicians' -GroupScope Global
Add-ADGroupMember -Identity 'technicians' -Members 'ferdinand'
```



```
Administrator: Windows PowerShell
PS C:\> New-ADUser -SamAccountName 'Ferdinand' -Name 'Ferdinand' -Enabled $true -AccountPassword (Read-Host "Enter User Password" -AsSecureString)
Enter User Password: *****
PS C:\> New-ADGroup -Name 'Technicians' -GroupScope Global
PS C:\> Add-ADGroupMember -Identity 'technicians' -Members 'ferdinand'
PS C:\>
```

Figure 15-28

This Active Directory user and group will be used in the next section of this chapter. You've just witnessed the power of PowerShell remoting in action, as all of this was accomplished via implicit remoting and without the need to install the Remote Server Administration Tools or Active Directory cmdlets locally.

## Advanced session configurations

We have a few technicians that we want to be able to run PowerShell commands on our servers via PowerShell remoting, but we don't want to make them an admin and we also don't want them to be able to run all of the available cmdlets. So, we're going to set up a constrained endpoint for delegated administration on the servers that we want them to be able to access. This will allow us to delegate certain administrative tasks to our technicians, while limiting our security risks.

```
Invoke-Command -ComputerName sql01, web01 {
New-Item c:\ -Name scripts -ItemType directory -ErrorAction SilentlyContinue
New-PSSessionConfigurationFile -Path c:\Scripts\technicians.pssc -ExecutionPolicy 'RemoteSigned' ` 
-VisibleCmdlets Get-Process, Select-Object, Where-Object
$sdld = "O:NSG:BAD:P(A;;GA;;;BA)(A;;GR;;;IU)(A;;GXGR;` 
;;S-1-5-21-1903002483-2365998268-193331953-1448)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)"` 
Register-PSSessionConfiguration -Path c:\Scripts\technicians.pssc -Name Technicians ` 
-SecurityDescriptorSddl $sdld
}
```

The screenshot shows an Administrator Windows PowerShell window. The command history at the top is:

```
PS C:\> Invoke-Command -ComputerName sql01, web01 {`n>> New-Item c:\ -Name scripts -ItemType directory -ErrorAction SilentlyContinue`n>> New-PSSessionConfigurationFile -Path c:\Scripts\technicians.pssc -ExecutionPolicy 'RemoteSigned' `n-VisibleCmdlets Get-Process, Select-Object, Where-Object`n>> $sdld = "O:NSG:BAD:P(A;;GA;;;BA)(A;;GR;;;IU)(A;;GXGR;`n`n;;S-1-5-21-1903002483-2365998268-193331953-1448)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)"`n>> Register-PSSessionConfiguration -Path c:\Scripts\technicians.pssc -Name Technicians `n-SecurityDescriptorSddl $sdld`n}`n`n
```

The output shows the creation of a 'scripts' directory and the registration of a session configuration named 'Technicians' with the specified SDDL. A warning message about restarting WinRM services is displayed twice.

Figure 15-29

We have to admit that learning Security Description Definition Language (SDDL) is difficult and if it's something you'd rather not invest the time in learning, you can use the **-ShowSecurityDescriptorUI** parameter (which will result in a GUI pop-up to specify the security on the custom endpoint) instead of the **-SecurityDescriptorSddl** parameter.

```
Invoke-Command -ComputerName sql01, web01 {
New-Item c:\ -Name scripts -ItemType directory -ErrorAction SilentlyContinue
New-PSSessionConfigurationFile -Path c:\Scripts\technicians.pssc -ExecutionPolicy 'RemoteSigned' ` 
-VisibleCmdlets Get-Process, Select-Object, Where-Object
Register-PSSessionConfiguration -Path c:\Scripts\technicians.pssc -Name Technicians ` 
-ShowSecurityDescriptorUI
}
```

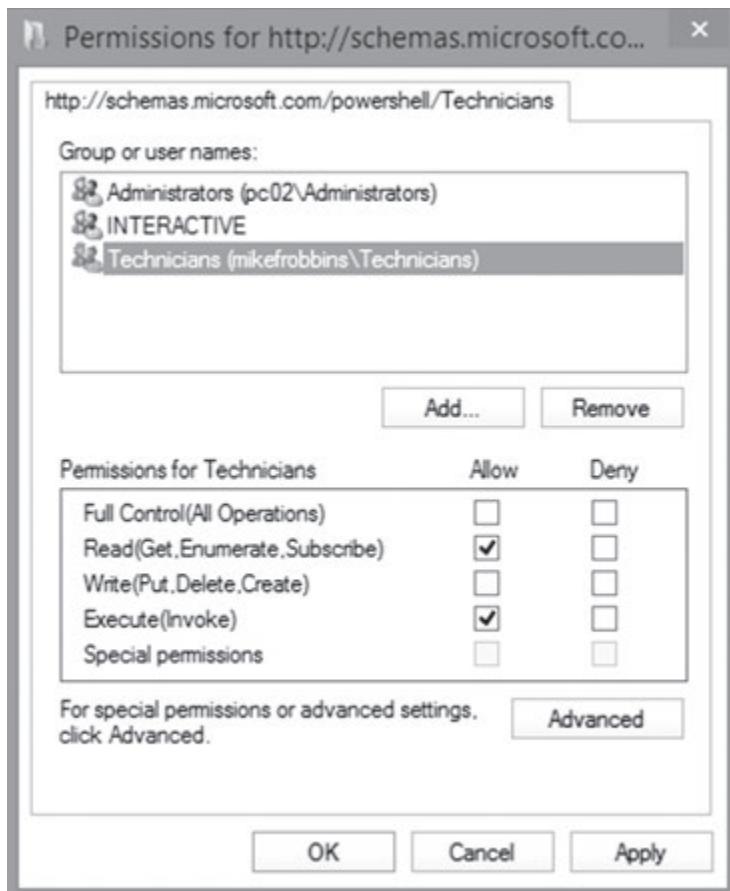


Figure 15-30

We've opened PowerShell as the user Ferdinand, who can now create a PSSession to our sql01 and web01 servers. The custom endpoint that we created is specified via the **-ConfigurationName** parameter, when establishing the PSSession.

```
$env:USERNAME
$session = New-PSSession -ComputerName sql01, web01 -ConfigurationName technicians
```

```
PS C:\> $env:USERNAME
ferdinand
PS C:\> $session = New-PSSession -ComputerName sql01, web01 -ConfigurationName technicians
PS C:\>
```

Figure 15-31

Ferdinand can now keep an eye on the processes on these servers.

```
Invoke-Command -Session $session { Get-Process | Where-Object PM -gt 10MB }
```

A screenshot of a Windows PowerShell window titled "Windows PowerShell". The command run is "PS C:\> Invoke-Command -Session \$session {Get-Process | Where-Object PM -gt 10MB} | Format-Table". The output is a table with columns: Handles, NPM(K), PM(K), WS(K), VM(M), CPU(s), Id, ProcessName, and PSComputerName. The table lists processes from two servers: sql01 and web01. The processes include SQL Agent, SQL Server, svchost, wsmprovhost, and IIS worker processes.

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName	PSComputerName
306	40	12336	4784	507		1580	SQLAGENT	sql01
658	60	250376	146924	1137		924	sqlservr	sql01
942	37	11628	21792	112		776	svchost	sql01
364	24	44904	63064	578	3.67	3032	wsmprovhost	sql01
933	37	10432	20768	111		768	svchost	web01
244	49	19672	13692	514		1068	IISv4	web01
570	25	49844	67748	578	3.27	1936	wsmprovhost	web01

Figure 15-32

Note, that he can't even use the **Format-List** cmdlet because we didn't give him access to use it.

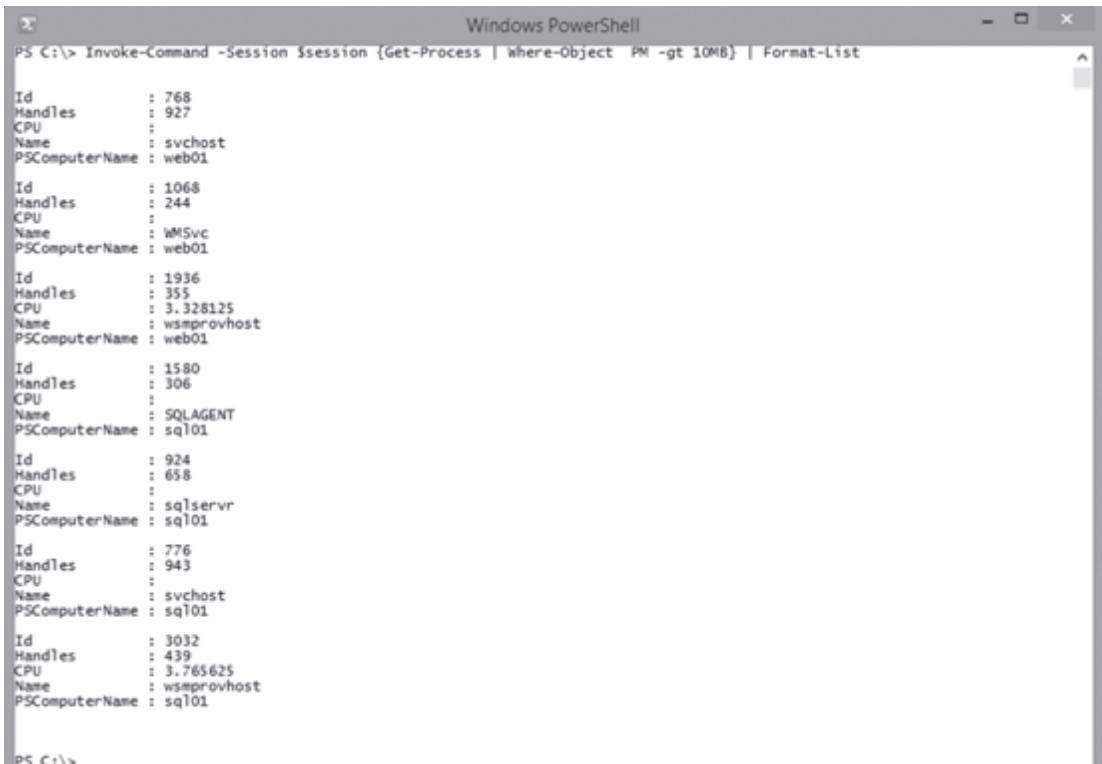
```
Invoke-Command -Session $session { Get-Process | Where-Object PM -gt 10MB | Format-List }
```

A screenshot of a Windows PowerShell window titled "Windows PowerShell". The command run is "PS C:\> Invoke-Command -Session \$session {Get-Process | Where-Object PM -gt 10MB | Format-List} | Format-Table". The output shows an error message: "The term 'Format-List' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again." Below the error message, the command "Format-Table" is shown again with the same error message. The command "Format-List" is also shown with the same error message.

Figure 15-33

What he could do though, if he wants to display a list view, is to pipe the results to **Format-List** after they're returned to his computer, since the restrictions we've put in place on what cmdlets he can run only applies to running remote commands on the servers.

```
Invoke-Command -Session $session { Get-Process | Where-Object PM -gt 10MB } | Format-List
```



```
Windows PowerShell
PS C:\> Invoke-Command -Session $session {Get-Process | where-Object PM -gt 10MB} | Format-List

Id          : 768
Handles    : 927
CPU        :
Name       : svchost
PSComputerName : web01

Id          : 1068
Handles    : 244
CPU        :
Name       : wMSvc
PSComputerName : web01

Id          : 1936
Handles    : 355
CPU        : 3.328125
Name       : wsmprovhost
PSComputerName : web01

Id          : 1580
Handles    : 306
CPU        :
Name       : SQLAGENT
PSComputerName : sq101

Id          : 924
Handles    : 658
CPU        :
Name       : sqlservr
PSComputerName : sq101

Id          : 776
Handles    : 943
CPU        :
Name       : svchost
PSComputerName : sq101

Id          : 3032
Handles    : 439
CPU        : 3.765625
Name       : wsmprovhost
PSComputerName : sq101

PS C:\>
```

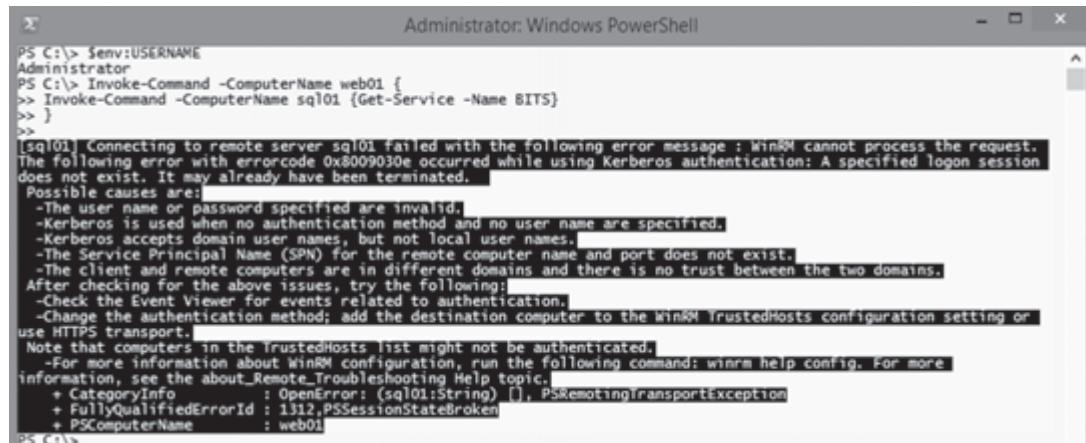
Figure 15-34

## Handling multi-hop remoting

When running a command on a remote server via PowerShell remoting, your credentials are delegated for that session. However, if you attempt to access another remote system from the system that you are remoted into, you'll receive an error because by default, your credentials can't be delegated by the remote machine to another remote machine. This is also known as the multi-hop or second-hop issue.

In Figure 15-35, we're back to running PowerShell as an admin and we use the **Invoke-Command** cmdlet to run a command on our web01 server. The command we're running on web01 is attempting to establish a remote session to sq101 to run a command. This generates an error because web01 can't delegate our credentials to the sq101 server for authentication.

```
$env:USERNAME
Invoke-Command -ComputerName web01 {
Invoke-Command -ComputerName sq101 { Get-Service -Name BITS }
}
```



```

PS C:\> $env:USERNAME
Administrator
PS C:\> Invoke-Command -ComputerName web01 {
>>   Invoke-Command -ComputerName sql01 {Get-Service -Name BITS}
>> }
>>
[sql01] Connecting to remote server sql01 failed with the following error message : WinRM cannot process the request.
The following error with errorcode 0x8009030e occurred while using Kerberos authentication: A specified logon session does not exist. It may already have been terminated.
Possible causes are:
-The user name or password specified are invalid.
-Kerberos is used when no authentication method and no user name are specified.
-Kerberos accepts domain user names, but not local user names.
-The Service Principal Name (SPN) for the remote computer name and port does not exist.
-The client and remote computers are in different domains and there is no trust between the two domains.
After checking for the above issues, try the following:
-Check the Event Viewer for events related to authentication.
-Change the authentication method; add the destination computer to the WinRM TrustedHosts configuration setting or use HTTPS transport.
Note that computers in the TrustedHosts list might not be authenticated.
-For more information about WinRM configuration, run the following command: winrm help config. For more information, see the about_Remote_Troubleshooting Help topic.
+ CategoryInfo          : OpenError: (sql01:String) [], PSRemotingTransportException
+ FullyQualifiedErrorId : 1312, PSSessionStateBroken
+ PSComputerName         : web01
PS C:\>

```

Figure 15-35

Although we've been logged into our Windows 8.1 workstation as an account with domain admin privileges for the sake of simplicity, in our opinion it's a best practice to log into your workstation as a user and if you are going to log in as a local admin, that account definitely shouldn't be a domain admin.

When you run PowerShell, it needs to be run as a local admin, regardless of whether you've logged into your PC as a user or local admin, because PowerShell is unable to participate in User Access Control. In our opinion, you should never run PowerShell as a domain admin—or even with an account that has elevated privileges in your domain. That means you'll be using the **-Credential** parameter when establishing PSSessions or by using the **Invoke-Command** cmdlet, because the account that you're running PowerShell as won't have access to accomplish the tasks on the servers that you're attempting to perform them on.

Start out by storing your elevated credentials in a variable.

```
$cred = Get-Credential
```

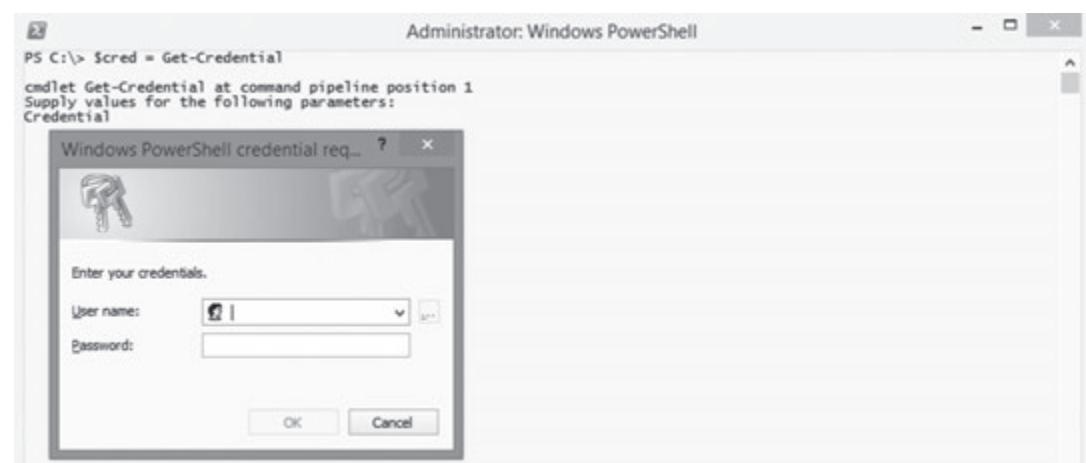


Figure 15-36

Running a command by using the **Invoke-Command** cmdlet or by using a PSSession will be accomplished, as shown in Figure 15-37. Another benefit to pre-creating a PSSession is that the **-Credential** parameter is only specified once and all of the commands that you run across that session will use the credentials that were used to establish the session.

```
Invoke-Command -ComputerName sql01 { Get-Service -Name BITS } -Credential $cred
$session = New-PSSession -ComputerName sql01 -Credential $cred
Invoke-Command -Session $session { Get-Service -Name BITS }
```

The screenshot shows an Administrator Windows PowerShell window. The commands entered are:

```
PS C:\> Invoke-Command -ComputerName sql01 { Get-Service -Name BITS } -Credential $cred
Status      Name          DisplayName          PSComputerName
----      ----          -----          -----
Stopped    BITS          Background Intelligent Transfer Ser... sql01

PS C:\> $session = New-PSSession -ComputerName sql01 -Credential $cred
PS C:\> Invoke-Command -Session $session {Get-Service -Name BITS}
Status      Name          DisplayName          PSComputerName
----      ----          -----          -----
Stopped    BITS          Background Intelligent Transfer Ser... sql01
```

Figure 15-37

What does all this have to do with multi-hop remoting? We believe that the easiest way to resolve any problem is to eliminate the problem altogether. We're going to resolve the multi-hop remoting issue without having to make any changes to the machines in your environment or reduce security in any way.

**\$Using** is a variable scope modifier that was introduced in PowerShell version 3 and it allows you to use a local variable in a remote session.

Since you will be storing the credentials in a variable that you will run remote commands as anyway, we'll also use the **\$Using** variable scope modifier to specify that same variable in the command that runs on the remote machine to authenticate to the third machine, thus eliminating the credential delegation problem altogether.

```
$cred = Get-Credential
Invoke-Command web01 {
    Invoke-Command -ComputerName sql01 { Get-Service -Name BITS } -Credential $Using:cred
} -Credential $cred
```

```

Administrator: Windows PowerShell
PS C:\> Scred = Get-Credential
cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
PS C:\> Invoke-Command -ComputerName web01 {
>> Invoke-Command -ComputerName sql01 {Get-Service -Name BITS} -Credential $Using:cred
>>} -Credential $cred
>>
Status      Name            DisplayName          PSComputerName
----      ----            ----          -----
Stopped    BITS           Background Intelligent Transfer Ser... web01

PS C:\> 

```

Figure 15-38

We could also enable the Credential Security Support Provider (CredSSP) protocol, which would allow our credentials to be delegated by the remote machine to the third computer in the chain that we're attempting to communicate with. First, we'll enable CredSSP on our Windows 8.1 workstation, which is named pc02, specifying the machine name of web01 as the value of the **-DelegateComputer** parameter, which is the name of the remote server that we'll be granting the ability to delegate our credentials.

```
Enable-WSManCredSSP -DelegateComputer web01 -Role Client
```

```

Administrator: Windows PowerShell
PS C:\> Enable-WSManCredSSP -DelegateComputer web01 -Role Client
CredSSP Authentication Configuration for WS-Management
CredSSP authentication allows the user credentials on this computer to be sent to a remote computer. If you use CredSSP authentication for a connection to a malicious or compromised computer, that computer will have access to your user name and password. For more information, see the Enable-WSManCredSSP Help topic.
Do you want to enable CredSSP authentication?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y

cfg      : http://schemas.microsoft.com/wbem/wsman/1/config/client/auth
Tang    : en-05
Basic   : true
Digest  : true
Kerberos: true
Negotiate: true
Certificate: true
CredSSP  : true

PS C:\> 

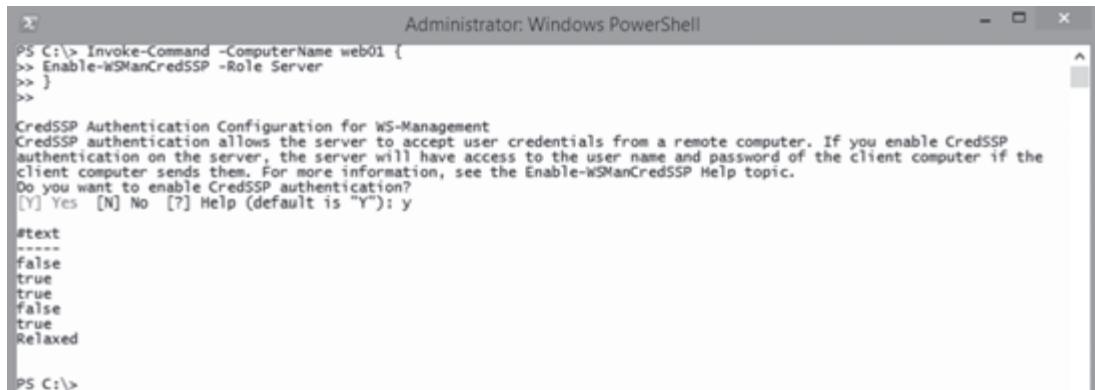
```

Figure 15-39

Next, we need to enable CredSSP on the web01 server itself.

```
Invoke-Command -ComputerName web01 {
  Enable-WSManCredSSP -Role Server
}
```

## Windows PowerShell: TFM



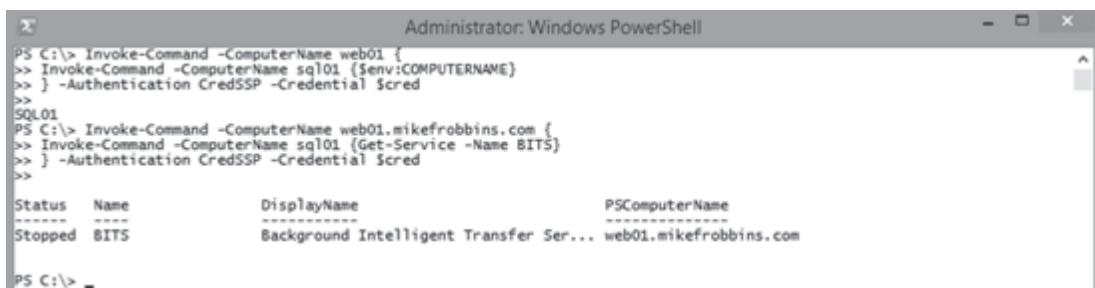
```
Administrator: Windows PowerShell
PS C:\> Invoke-Command -ComputerName web01 {
>> Enable-WsManCredSSP -Role Server
>> }
>>
CredSSP Authentication Configuration for WS-Management
CredSSP authentication allows the server to accept user credentials from a remote computer. If you enable CredSSP
authentication on the server, the server will have access to the user name and password of the client computer if the
client computer sends them. For more information, see the Enable-WsManCredSSP Help topic.
Do you want to enable CredSSP authentication?
[Y] Yes [N] No [?] Help (default is "Y"): y
#text
-----
false
true
true
false
true
Relaxed
PS C:\>
```

Figure 15-40

Now, we'll run a couple of sets of commands just to make sure everything is working as expected.

```
Invoke-Command -ComputerName web01 {
    Invoke-Command -ComputerName sql01 {$env:COMPUTERNAME}
} -Authentication CredSSP -Credential $cred

Invoke-Command -ComputerName web01 {
    Invoke-Command -ComputerName sql01 {Get-Service -Name BITS}
} -Authentication CredSSP -Credential $cred
```



```
Administrator: Windows PowerShell
PS C:\> Invoke-Command -ComputerName web01 {
>> Invoke-Command -ComputerName sql01 {$env:COMPUTERNAME}
>> } -Authentication CredSSP -Credential $cred
>>
SQL01
PS C:\> Invoke-Command -ComputerName web01.mikefrobbins.com {
>> Invoke-Command -ComputerName sql01 {Get-Service -Name BITS}
>> } -Authentication CredSSP -Credential $cred
>>
Status   Name          DisplayName          PSComputerName
-----  --  -----
Stopped  BITS          Background Intelligent Transfer Ser...  web01.mikefrobbins.com
PS C:\> _
```

Figure 15-41

The name of the computer specified via the **-DelegateComputer** parameter when enabling CredSSP on the workstation must match the name that you use to connect to the remote machine. If you specify a fully qualified domain name, then you need to use that same fully qualified domain name when establishing a PSSession or by using the **-ComputerName** parameter of **Invoke-Command**. Wildcards are valid. However, if you're using a wildcard, limit the scope to your domain name such as \*.sapien.com.

## Exercise 15 — PowerShell remoting

You've been promoted to the Team Lead of the Server Admins and you've been tasked with implementing a mechanism for performing remote administration of servers. The requirements of the solution are that the cost be kept at an absolute minimum and it's preferable not to have to install any type of additional software on the servers, although existing software and settings may be modified as needed.

You're in luck! All of the desktops and servers that you are responsible for supporting are new enough that they at least have PowerShell version 2 or higher installed, which means that you can use PowerShell remoting!

### Task 1

Enable PowerShell remoting on all the machines in your test lab environment. If your test lab is your workstation at your place of employment, check with your supervisor before completing this step and continuing.

### Task 2

Create a one-to-one interactive PowerShell remoting session to one of the machines in your test environment. If you only have a single computer, use **\$Env:ComputerName** as the remote computer name to establish a session to the local computer.

Determine what cmdlet is used to list the PowerShell modules that are installed, and then run that cmdlet in the PowerShell one-to-one session to see what modules are installed on the remote computer. Here's a small hint: What is the singular form of the word "modules"? That is the noun portion of the cmdlet you need.

### Task 3

This time, use one-to-many remoting to query a list of the installed modules on one or more remote machines in your test environment.

Locate a cmdlet to compare those results to your local machine; that is unless you're only working with a single machine. The verb for the cmdlet that you need to perform the comparison is "compare."

### Task 4

Create a PSSession to a remote computer or **\$Env:ComputerName** if you're working with a single machine. Disconnect the PSSession and reconnect to it.

Use **Invoke-Command** with the PSSession to query the date and time on the remote machine(s) to verify they're set correctly.



## Chapter 16

# Background and scheduled jobs

### What's a job?

A job is how you earn a living and a PowerShell job is what we consider the ultimate way to make a living, because you're doing nothing but scripting in PowerShell all day, every day. Oh, you meant what are jobs in PowerShell.

Once started, long-running commands in PowerShell monopolize your session by making you wait, while you sit at a flashing cursor, until the command completes.

```
Start-Sleep -Seconds 90
```



Figure 16-1

After completion, your prompt is returned and you can move onto running another command. We don't know about you, but this doesn't sound like an efficient use of time. Jobs in PowerShell are the answer, so that our PowerShell session isn't tied up for the duration of each command. Jobs are an efficient way

for us to give PowerShell some commands and have PowerShell do its thing in the background and “make it so,” immediately giving us our prompt back so that we can continue working, without waiting for the results of the job.

There are a couple of different ways to run a command as a job in PowerShell. Some cmdlets have built-in native support for running as a job and these cmdlets have an **-AsJob** parameter. Other cmdlets, such as the **Start-Sleep** cmdlet that was used in Figure 16-1, don’t have an **-AsJob** parameter. Those cmdlets can be run as a job, by using the **Start-Job** cmdlet.

## Local background jobs

We’re going to run two commands inside the script block of the **Start-Job** cmdlet and as with other cmdlets that have a **-ScriptBlock** parameter, there’s no reason to use a semicolon to separate the different commands, simply use a line break and format your code for readability.

```
Start-Job -ScriptBlock {
    Get-WindowsOptionalFeature -Online
    Start-Sleep -Seconds 90
}
```

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
2	Job2	BackgroundJob	Running	True	localhost	

Figure 16-2

Notice the type of job is a background job, as shown in Figure 16-2.

How does this work, you might ask? Before the previous command was started, there was one instance of PowerShell running on our workstation.

```
Get-Process -Name PowerShell
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
379	24	66304	71884	609	1.36	3012	powershell

Figure 16-3

When we ran the **Start-Job** cmdlet, a second instance of PowerShell was launched in the background and our commands that are contained in the **ScriptBlock** of **Start-Job** are run using this background instance of PowerShell.

```
Get-Process -Name PowerShell
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
392	29	53504	72540	619	0.47	460	powershell
480	31	80636	86640	685	2.00	3012	powershell

Figure 16-4

## Remote background jobs

There are a number of commands in PowerShell that natively support being run as a job through the use of the **-AsJob** parameter. Here's how to find a list of the commands that are included in PowerShell version 4.

```
Get-Command -ParameterName AsJob
```

CommandType	Name	ModuleName
Cmdlet	Get-WmiObject	Microsoft.PowerShell.Management
Cmdlet	Invoke-Command	Microsoft.PowerShell.Core
Cmdlet	Invoke-WmiMethod	Microsoft.PowerShell.Management
Cmdlet	Remove-WmiObject	Microsoft.PowerShell.Management
Cmdlet	Restart-Computer	Microsoft.PowerShell.Management
Cmdlet	Set-WmiInstance	Microsoft.PowerShell.Management
Cmdlet	Stop-Computer	Microsoft.PowerShell.Management
Cmdlet	Test-Connection	Microsoft.PowerShell.Management

Figure 16-5

There aren't very many commands that have an **-AsJob** parameter, when you consider that we have the Remote Server Administrator Tools installed on our Windows 8.1 client machine that we're running these commands on. What about all of the other cmdlets?

One of the problems is that we've all been spoiled by the new module auto-loading feature that was introduced in PowerShell version 3. If we first import all of our PowerShell modules that are installed on the local machine, you'll notice that there are a lot more cmdlets that have an **-AsJob** parameter. This is because searching for cmdlets that have a specific parameter doesn't trigger the automatic loading of modules. We've also specified the **-DisableNameChecking** parameter in the following command,

because we have the `SQLPS` PowerShell module installed, which is used for managing SQL Server with PowerShell. If that parameter wasn't specified, we'd receive a warning about some of the cmdlets in that module that use unapproved verbs.

```
Get-Module -ListAvailable | Import-Module -DisableNameChecking
Get-Command -ParameterName AsJob
```

CommandType	Name	ModuleName
Alias	Export-DnsServerTrustAnchor	DnsServer
Alias	Flush-Volume	Storage
Alias	Get-VpnServerIPsecConfiguration	RemoteAccess
Alias	Initialize-Volume	Storage
Alias	Move-SmbClient	SmbWitness
Alias	Reconcile-DhcpServerv4IPRecord	DhcpServer
Alias	Set-VpnServerIPsecConfiguration	RemoteAccess
Alias	Write-FileSystemCache	Storage
Function	Add-BcDataCacheExtension	BranchCache
Function	Add-BgpCustomRoute	RemoteAccess
Function	Add-BgpPeer	RemoteAccess
Function	Add-BgpRouter	RemoteAccess
Function	Add-BgpRoutingPolicy	RemoteAccess
Function	Add-BgpRoutingPolicyForPeer	RemoteAccess
Function	Add-DaaAppServer	RemoteAccess
Function	Add-DaClient	RemoteAccess
Function	Add-DaClientDnsConfiguration	RemoteAccess
Function	Add-DaEntryPoint	RemoteAccess
Function	Add-DaVmServer	RemoteAccess
Function	Add-DhcpServerInDC	DhcpServer
Function	Add-DhcpServerSecurityGroup	DhcpServer
Function	Add-DhcpServerv4Class	DhcpServer
Function	Add-DhcpServerv4ExclusionRange	DhcpServer
Function	Add-DhcpServerv4Failover	DhcpServer
Function	Add-DhcpServerv4FailoverScope	DhcpServer
Function	Add-DhcpServerv4Filter	DhcpServer
Function	Add-DhcpServerv4Lease	DhcpServer
Function	Add-DhcpServerv4MulticastExclusionRange	DhcpServer
Function	Add-DhcpServerv4MulticastScope	DhcpServer
Function	Add-DhcpServerv4OptionDefinition	DhcpServer
Function	Add-DhcpServerv4Policy	DhcpServer

Figure 16-6

Now that we've imported all of the modules on our workstation, we can now see that there are actually a total of 1086 commands that have an `-AsJob` parameter, but keep in mind that many of those are functions and if you happen to be using Windows 7, even with PowerShell version 3 or 4 loaded, you won't have all of these commands.

```
(Get-Command -ParameterName AsJob).Count
```

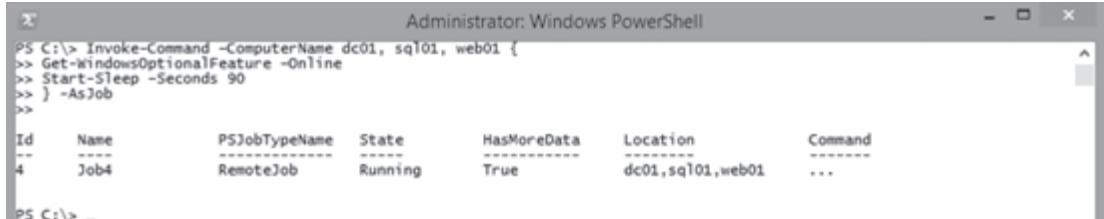
```
PS C:\> (Get-Command -ParameterName AsJob).Count
1086
PS C:\>
```

Figure 16-7

We'll run a remote job against three servers, by using the PowerShell remoting `Invoke-Command` cmdlet and specifying the same commands inside our script block that we used in our local job. As shown in Figure 16-8, the `-AsJob` parameter is used to run this command as remote jobs on the three servers.

## Background and scheduled jobs

```
Invoke-Command -ComputerName dc01, sql01, web01 {  
    Get-WindowsOptionalFeature -Online  
    Start-Sleep -Seconds 90  
} -AsJob
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Invoke-Command -ComputerName dc01, sql01, web01 {  
    >> Get-WindowsOptionalFeature -Online  
    >> Start-Sleep -Seconds 90  
    >> } -AsJob  
>>
```

Below the command, a table displays the job information:

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
4	Job4	RemoteJob	Running	True	dc01,sql01,web01	...

PS C:\> \_

Figure 16-8

Notice in Figure 16-8 that this job type is a remote job, which differs from the job created in the previous section, which was a background job.

## Managing jobs

OK, so we've run these jobs, but how do we retrieve the results of the commands we ran and how do we find out if they ran successfully? The **Get-Job** cmdlet is used to check on the status of whether or not the job is still running, completed, or if it failed.

```
Get-Job
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

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

Below the command, a table displays the job information:

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
2	Job2	BackgroundJob	Completed	True	localhost	...
4	Job4	RemoteJob	Running	True	dc01,sql01,web01	...

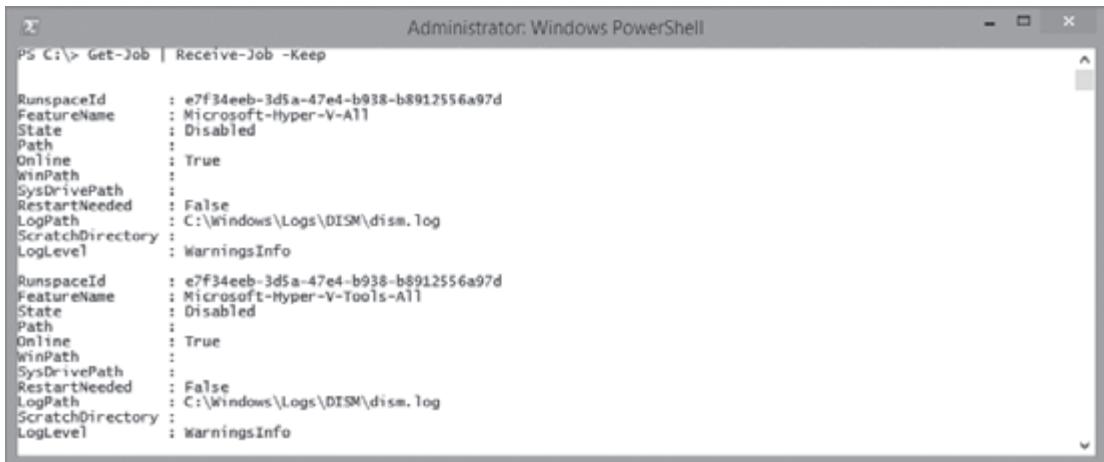
PS C:\>

Figure 16-9

To retrieve the results of all of our jobs, we'll find all of the jobs and pipe them to the **Receive-Job** cmdlet and specify the **-Keep** parameter.

```
Get-Job | Receive-Job -Keep
```

## Windows PowerShell: TFM



```
Administrator: Windows PowerShell
PS C:\> Get-Job | Receive-Job -Keep

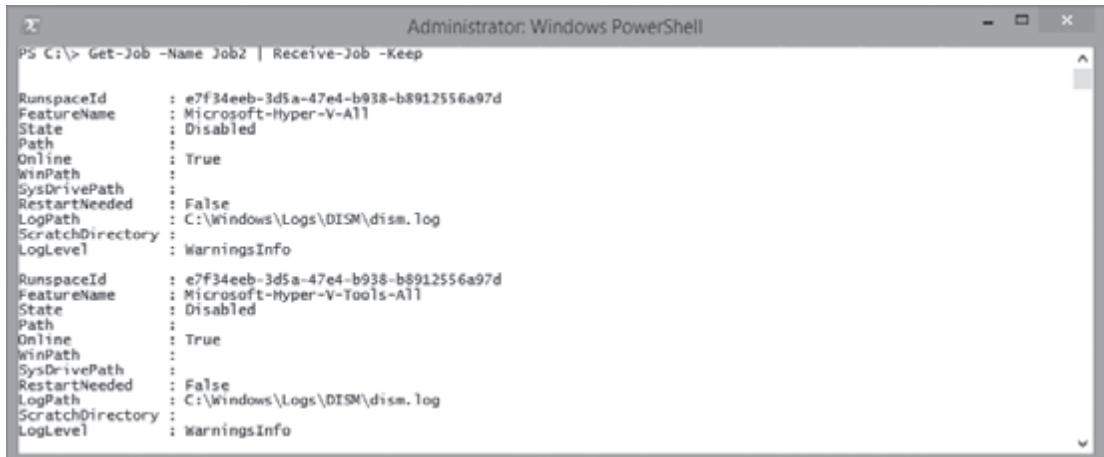
RunspaceId      : e7f34eeb-3d5a-47e4-b938-b8912556a97d
FeatureName     : Microsoft-Hyper-V-All
State           : Disabled
Path             :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo

RunspaceId      : e7f34eeb-3d5a-47e4-b938-b8912556a97d
FeatureName     : Microsoft-Hyper-V-Tools-All
State           : Disabled
Path             :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo
```

Figure 16-10

By piping the results of **Get-Job** to **Receive-Job**, we received the result of all of our jobs. To receive the results of a specific job, filter the results down with **Get-Job**, before piping to **Receive-Job**.

```
Get-Job -Name Job2 | Receive-Job -Keep
```



```
Administrator: Windows PowerShell
PS C:\> Get-Job -Name Job2 | Receive-Job -Keep

RunspaceId      : e7f34eeb-3d5a-47e4-b938-b8912556a97d
FeatureName     : Microsoft-Hyper-V-All
State           : Disabled
Path             :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo

RunspaceId      : e7f34eeb-3d5a-47e4-b938-b8912556a97d
FeatureName     : Microsoft-Hyper-V-Tools-All
State           : Disabled
Path             :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo
```

Figure 16-11

You could also use one of the parameters of **Receive-Job** to filter the results down, without piping the results of **Get-Job** to **Receive-Job**.

```
Receive-Job -Name Job2 -Keep
```

## Background and scheduled jobs

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Receive-Job -Name Job2 -Keep was run. The output displays two sets of properties for jobs, each with RunspaceId, FeatureName, State, Path, Online, WinPath, SysDrivePath, RestartNeeded, LogPath, ScratchDirectory, and LogLevel.

```
PS C:\> Receive-Job -Name Job2 -Keep

RunspaceId      : e7f34eeb-3d5a-47e4-b938-b8912556a97d
FeatureName     : Microsoft-Hyper-V-All
State           : Disabled
Path            :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo

RunspaceId      : e7f34eeb-3d5a-47e4-b938-b8912556a97d
FeatureName     : Microsoft-Hyper-V-Tools-All
State           : Disabled
Path            :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo
```

Figure 16-12

It's also possible to filter and select only certain data from the results.

```
Receive-Job -Name Job4 -Keep | Where-Object FeatureName -eq TelnetClient |
Select-Object PsComputerName, State
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Receive-Job -Name Job4 -Keep | Where-Object FeatureName -eq TelnetClient | Select-Object PsComputerName, State was run. The output shows a table with PsComputerName and State columns.

PsComputerName	State
dc01	Disabled
sql01	Disabled
web01	Disabled

Figure 16-13

By specifying the **-Keep** parameter, as we did in Figure 16-13, you can retrieve the results again, but retrieving the results without specifying the **-Keep** parameter causes the buffer that holds the results to be cleared.

```
Get-Job -Name Job2 | Receive-Job
```

## Windows PowerShell: TFM

```
Administrator: Windows PowerShell
PS C:\> Get-Job -Name Job2 | Receive-Job

RunspaceId      : e7f34eeb-3d5a-47e4-b938-b8912556a97d
FeatureName     : Microsoft-Hyper-V-All
State           : Disabled
Path            :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo

RunspaceId      : e7f34eeb-3d5a-47e4-b938-b8912556a97d
FeatureName     : Microsoft-Hyper-V-Tools-All
State           : Disabled
Path            :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo
```

Figure 16-14

We'll now attempt to receive the results of **Job2** again.

```
Get-Job -Name Job2 | Receive-Job
```

```
Administrator: Windows PowerShell
PS C:\> Get-Job -Name Job2 | Receive-Job
PS C:\>
```

Figure 16-15

Notice that in Figure 16-15, there aren't any results. That's because we already retrieved them without using the **-Keep** parameter, which deleted them and prevents them from being retrieved again.

Even after the results have been retrieved, the job itself still exists. Notice that **Job2** now has **False** in the **HasMoreData** column, when we run the **Get-Job** cmdlet.

```
Get-Job
```

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
--	---	-----	-----	-----	-----	-----
2	Job2	BackgroundJob	Completed	False	localhost	...
4	Job4	RemoteJob	Completed	True	dc01,sq101,web01	...

Figure 16-16

The **Remove-Job** cmdlet is used to remove the job.

```
Get-Job -Name Job2 | Remove-Job
Get-Job
```

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
4	Job4	RemoteJob	Completed	True	dc01,sql01,web01	...

Figure 16-17

The **Receive-Job** cmdlet has an **-AutoRemoveJob** parameter that does just what it says—it will automatically remove the job when the results are retrieved. There's a caveat to the **-AutoRemoveJob** parameter though—it's only valid when used with the **-Wait** parameter. The **-Wait** parameter suppresses the command prompt until the job is complete. So if this command is run on a job that's going to take thirty more minutes to complete, then you're back to waiting on your prompt, just as if the commands weren't run as a job.

```
Get-Job
Receive-Job -Id 4 -Wait -AutoRemoveJob
```

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
4	Job4	RemoteJob	Completed	True	dc01,sql01,web01	...

```
PS C:\> Receive-Job -Id 4 -Wait -AutoRemoveJob

PSComputerName : dc01
RunspaceId : 86e86c7e-4109-46f4-a569-21b69b7f3a40
FeatureName : NetFx4ServerFeatures
State : Enabled
Path :
Online : True
WinPath :
SysDrivePath :
RestartNeeded : False
LogPath : C:\Windows\Logs\DISM\dism.log
ScratchDirectory :
LogLevel : WarningsInfo

PSComputerName : dc01
RunspaceId : 86e86c7e-4109-46f4-a569-21b69b7f3a40
FeatureName : NetFx4
State : Enabled
Path :
Online : True
WinPath :
```

Figure 16-18

To prevent this problem when using the **-AutoRemoveJob** parameter, use the **-State** parameter of **Get-Job** to filter down to only the completed jobs, pipe those results to **Receive-Job**, and then use the

**-AutoRemoveJob** parameter. This prevents the results from unfinished jobs from being retrieved and it will keep your prompt from being tied up until the jobs have completed and are automatically removed.

```
Get-Job -State Completed | Receive-Job -Wait -AutoRemoveJob
```

```
Administrator: Windows PowerShell
PS C:\> Get-Job -State Completed | Receive-Job -Wait -AutoRemoveJob

RunspaceId      : e091aba8-66ab-4e17-9b46-ae9f82d9888d
FeatureName     : Microsoft-Hyper-V-All
State           : Disabled
Path            :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo

RunspaceId      : e091aba8-66ab-4e17-9b46-ae9f82d9888d
FeatureName     : Microsoft-Hyper-V-Tools-All
State           : Disabled
Path            :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel         : WarningsInfo

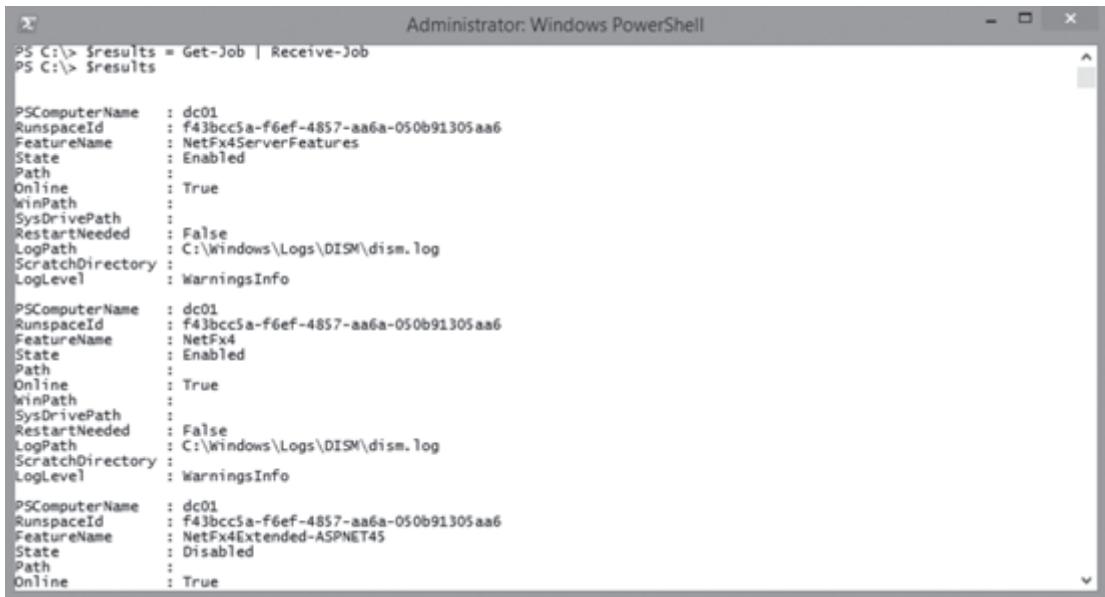
RunspaceId      : e091aba8-66ab-4e17-9b46-ae9f82d9888d
FeatureName     : Microsoft-Hyper-V
State           : Disabled
Path            :
Online          : True
WinPath          :
SysDrivePath    :
RestartNeeded   : False
LogPath          : C:\Windows\Logs\DISM\dism.log
```

Figure 16-19

## Storing output from jobs

If you would like to store the results of a job short term, either continue to use the **-Keep** parameter when retrieving its results or store the results in a variable.

```
$results = Get-Job | Retrieve-Job
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$results = Get-Job | Receive-Job was run, followed by PS C:\> \$results. The output displays three objects, each representing a job. Each object has properties: PSComputerName, RunspaceId, FeatureName, State, Path, Online, WinPath, SysDrivePath, RestartNeeded, LogPath, ScratchDirectory, and LogLevel. The first two objects have FeatureName set to "NetFx45ServerFeatures" and "NetFx4", respectively, while the third has "NetFx4Extended-ASPNET45". The State for all three is "Enabled". The Path is empty for all. Online is True for all. WinPath is present for the first two. SysDrivePath is present for the first two. RestartNeeded is False for the first two and True for the third. LogPath is C:\Windows\Logs\DISM\dism.log for all. ScratchDirectory is WarningsInfo for all. LogLevel is WarningsInfo for all.

```
PS C:\> $results = Get-Job | Receive-Job
PS C:\> $results

PSComputerName : dc01
RunspaceId     : f43bcc5a-f6ef-4857-aa6a-050b91305aa6
FeatureName    : NetFx45ServerFeatures
State          : Enabled
Path           :
Online         : True
WinPath        :
SysDrivePath   :
RestartNeeded  : False
LogPath        : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel       : WarningsInfo

PSComputerName : dc01
RunspaceId     : f43bcc5a-f6ef-4857-aa6a-050b91305aa6
FeatureName    : NetFx4
State          : Enabled
Path           :
Online         : True
WinPath        :
SysDrivePath   :
RestartNeeded  : False
LogPath        : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel       : WarningsInfo

PSComputerName : dc01
RunspaceId     : f43bcc5a-f6ef-4857-aa6a-050b91305aa6
FeatureName    : NetFx4Extended-ASPNET45
State          : Disabled
Path           :
Online         : True
```

Figure 16-20

To keep the results indefinitely, store them in a file. While the results could be stored in a simple text file and reimported into PowerShell, they would be stored and reimported as strings. Exporting them in CliXML format allows you to reimport them and work with them as deserialized objects, just as you would have worked with the original results, which were also deserialized objects.

```
Get-Job | Receive-Job | Export-Clixml -Path c:\scripts\jobresults.xml
Import-Clixml -Path C:\Scripts\jobresults.xml
```

```

Administrator: Windows PowerShell
PS C:\> Get-Job | Receive-Job | Export-Clixml -Path c:\scripts\jobresults.xml
PS C:\> Import-Clixml -Path C:\Scripts\jobresults.xml

PSComputerName : dc01
RunspaceId     : 911a666a-232a-409d-ba99-744acc5b0994
FeatureName    : NetFx4ServerFeatures
State          : Enabled
Path           :
Online         : True
WinPath        :
SysDrivePath   :
RestartNeeded  : False
LogPath        : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel       : WarningsInfo

PSComputerName : dc01
RunspaceId     : 911a666a-232a-409d-ba99-744acc5b0994
FeatureName    : NetFx4
State          : Enabled
Path           :
Online         : True
WinPath        :
SysDrivePath   :
RestartNeeded  : False
LogPath        : C:\Windows\Logs\DISM\dism.log
ScratchDirectory:
LogLevel       : WarningsInfo

PSComputerName : dc01
RunspaceId     : 911a666a-232a-409d-ba99-744acc5b0994
FeatureName    : NetFx4Extended-ASPNET45
State          : Disabled
Path           :
Online         : True

```

Figure 16-21

## Scheduled jobs

The ability to schedule jobs in PowerShell is a new feature that was introduced with PowerShell version 3. The cmdlets that we'll be discussing in this portion of the chapter are part of the **PSScheduledJob** module.

```
Get-Command -Module PSScheduledJob
```

CommandType	Name	ModuleName
Cmdlet	Add-JobTrigger	PSScheduledJob
Cmdlet	Disable-JobTrigger	PSScheduledJob
Cmdlet	Disable-ScheduledJob	PSScheduledJob
Cmdlet	Enable-JobTrigger	PSScheduledJob
Cmdlet	Enable-ScheduledJob	PSScheduledJob
Cmdlet	Get-JobTrigger	PSScheduledJob
Cmdlet	Get-ScheduledJob	PSScheduledJob
Cmdlet	Get-ScheduledJobOption	PSScheduledJob
Cmdlet	New-JobTrigger	PSScheduledJob
Cmdlet	New-ScheduledJobOption	PSScheduledJob
Cmdlet	Register-ScheduledJob	PSScheduledJob
Cmdlet	Remove-JobTrigger	PSScheduledJob
Cmdlet	Set-JobTrigger	PSScheduledJob
Cmdlet	Set-ScheduledJob	PSScheduledJob
Cmdlet	Set-ScheduledJobOption	PSScheduledJob
Cmdlet	Unregister-ScheduledJob	PSScheduledJob

Figure 16-22

Scheduled jobs are similar to PowerShell background jobs, except they can be scheduled to execute instead of having to run them manually. As with the jobs that we've previously discussed in this chapter, scheduled jobs run in the background and they can be managed with the **Job-** cmdlets that were covered in the "Managing Jobs" section of this chapter.

In addition to managing them with the **Job-** cmdlets, they can be viewed and managed outside of PowerShell, from within the Task Scheduler GUI (taskschd.msc) or the Schtasks.exe command-line utility.

Scheduled jobs themselves are saved to disk instead of only existing in memory and they live beyond the scope of your PowerShell session, which means that they exist even if you close and reopen your PowerShell console window or reboot your machine.

## Options for scheduled jobs

The **New-ScheduledTaskOption** cmdlet is used to set job options. This corresponds to various options that exist in several different locations in the task scheduler GUI. The **Run with highest privileges** option exists on the **General** tab of the GUI in the **Security options** section.

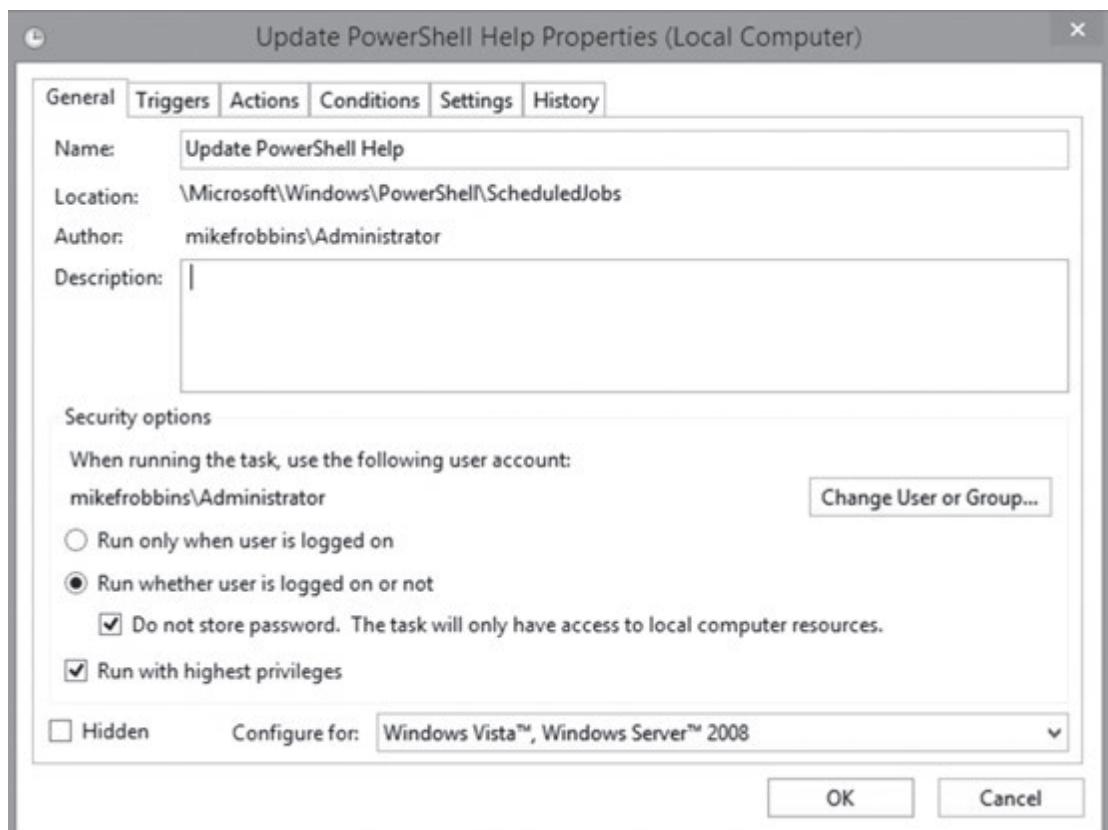


Figure 16-23

## Windows PowerShell: TFM

We're going to store our scheduled job options in a variable, so it's a little easier to use and clearer to understand what's going on when we use it later, instead of specifying everything with a one liner.

```
$JobOption = New-ScheduledJobOption -RunElevated
```



Figure 16-24

There are a number of different options that can be set with this cmdlet. The available parameters for it are shown in Figure 16-25.

```
(Get-Command -Name New-ScheduledJobOption | Select-Object -ExpandProperty parameters).keys
```

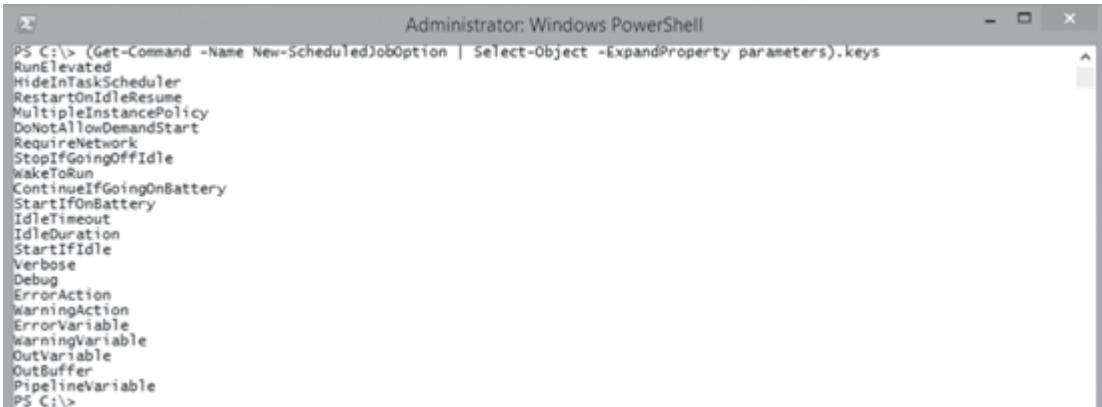


Figure 16-25

Later, we'll register our scheduled job with the task scheduler and once that's complete, you will be able to view the options that were set by using the **Get-ScheduledJobOption** cmdlet.

```
Get-ScheduledJob -Name 'Update PowerShell Help' | Get-ScheduledJobOption
```

## Background and scheduled jobs

```
Administrator: Windows PowerShell
PS C:\> Get-ScheduledJob -Name 'Update PowerShell Help' | Get-ScheduledJobOption
StartIfOnBatteries      : False
StopIfGoingOnBatteries   : True
WakeToRun                : False
StartIfNotIdle           : True
StopIfGoingOffIdle       : False
RestartOnIdleResume     : False
IdleDuration             : 00:10:00
IdleTimeout              : 01:00:00
ShowInTaskScheduler       : True
RunElevated               : True
RunWithoutNetwork         : True
DoNotAllowDemandStart    : False
MultipleInstancePolicy   : IgnoreNew
JobDefinition             : Microsoft.PowerShell.ScheduledJob.ScheduledJobDefinition
PS C:\> _
```

Figure 16-26

We often see users delete items that are created because they don't know how to modify them. It's simply easier for those users to delete and recreate those items, instead of learning how to modify them. For that reason, we're going to teach you how to modify these settings.

To modify one of these settings after your scheduled job has been registered, pipe the previous command to **Set-ScheduledJobOption** and specify the necessary changes. Specify the **-PassThru** parameter to see the results without having to use a **Get-** cmdlet to retrieve them.

```
Get-ScheduledJob -Name 'Update PowerShell Help' | Get-ScheduledJobOption |
Set-ScheduledJobOption -RequireNetwork -PassThru
```

```
Administrator: Windows PowerShell
PS C:\> Get-ScheduledJob -Name 'Update PowerShell Help' | Get-ScheduledJobOption |
>> Set-ScheduledJobOption -RequireNetwork -PassThru
>>
StartIfOnBatteries      : False
StopIfGoingOnBatteries   : True
WakeToRun                : False
StartIfNotIdle           : True
StopIfGoingOffIdle       : False
RestartOnIdleResume     : False
IdleDuration             : 00:10:00
IdleTimeout              : 01:00:00
ShowInTaskScheduler       : True
RunElevated               : True
RunWithoutNetwork         : False
DoNotAllowDemandStart    : False
MultipleInstancePolicy   : IgnoreNew
JobDefinition             : Microsoft.PowerShell.ScheduledJob.ScheduledJobDefinition
PS C:\> _
```

Figure 16-27

As shown in Figure 16-27, the job will only run if the network is available. We're working on setting up a scheduled job to update PowerShell's updatable Help on our workstation, which can't possibly complete without network access.

## Triggers for scheduled jobs

The **New-JobTrigger** cmdlet is used to define when the task will run. This corresponds to the **Triggers** tab in the GUI.

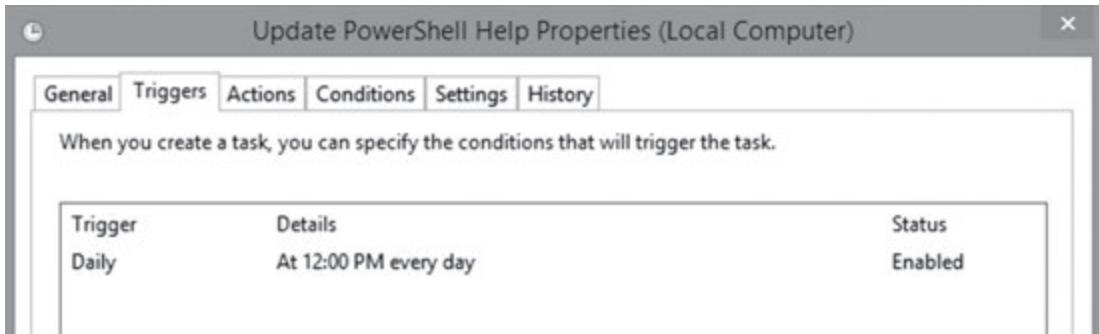


Figure 16-28

We want our job to execute at noon each day. We'll also store this information in a variable.

```
$NoonDailyTrigger = New-JobTrigger -Daily -At '12:00 PM'
```

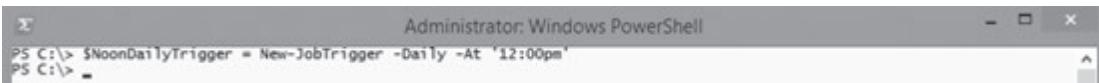


Figure 16-29

The command that we're about to run may seem like *déjà vu*, but that's one thing you'll learn to love about PowerShell. All PowerShell cmdlets are designed to look and feel consistent and you'll often find yourself thinking you've done this before, even when working with new cmdlets.

The job itself must be registered, which we'll cover later in this chapter, but once it is, you can view the triggers for a scheduled job by piping the **Get-ScheduledJob** cmdlet to **Get-JobTrigger**.

```
Get-ScheduledJob -Name 'Update PowerShell Help' | Get-JobTrigger
```

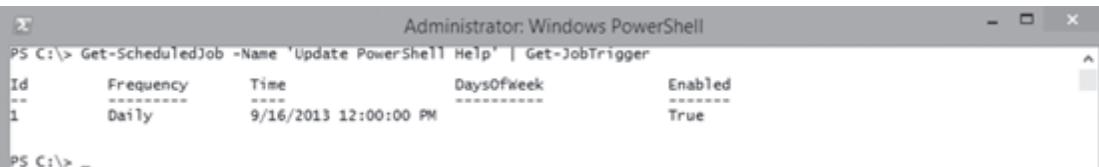


Figure 16-30

We'll pipe the previous command to **Set-JobTrigger** to modify it, because we've decided to run our task at 12:30, instead of noon, to make sure it runs while we're at lunch each day.

```
Get-ScheduledJob -Name 'Update PowerShell Help' | Get-JobTrigger |
Set-JobTrigger -At '12:30' -PassThru
```

```
Administrator: Windows PowerShell
PS C:\> Get-ScheduledJob -Name 'Update PowerShell Help' | Get-JobTrigger |
>> Set-JobTrigger -At '12:30pm' -PassThru
>>
Id      Frequency     Time          DaysOfWeek      Enabled
--      -----     ----          -----          -----
1       Daily        9/16/2013 12:30:00 PM    True
PS C:\>
```

Figure 16-31

There are also a number of different available parameters for this cmdlet and these can be used to set any of the options that you may have previously used when configuring scheduled tasks in the GUI.

```
(Get-Command -Name New-JobTrigger | Select -ExpandProperty parameters).keys
```

```
Administrator: Windows PowerShell
PS C:\> (Get-Command -Name New-JobTrigger | Select-Object -ExpandProperty parameters).keys
DaysInterval
WeeksInterval
RandomDelay
At
User
DaysOfWeek
AtStartup
AtLogOn
Once
RepetitionInterval
RepetitionDuration
RepeatIndefinitely
Daily
Weekly
Verbose
Debug
ErrorAction
WarningAction
ErrorVariable
WarningVariable
OutVariable
OutBuffer
PipelineVariable
PS C:\> _
```

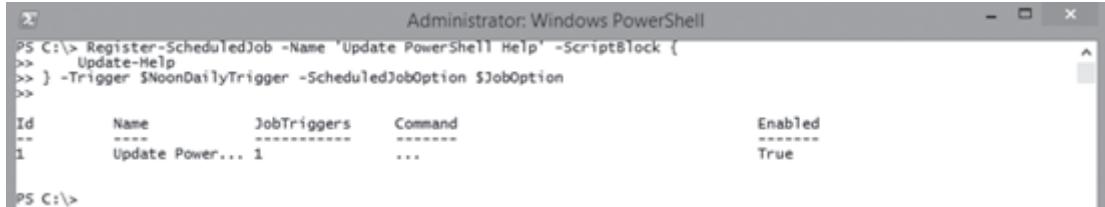
Figure 16-32

## Registering and using scheduled jobs

The **Register-ScheduledJob** cmdlet registers the job in the task scheduler, creates a folder hierarchy on disk, and stores the job's definition on disk. We'll now create our "Update PowerShell Help" scheduled job, by using the **Register-ScheduledJob** cmdlet and the variables that we used to store our trigger information and options.

## Windows PowerShell: TFM

```
Register-ScheduledJob -Name 'Update PowerShell Help' -ScriptBlock {  
    Update-Help  
} -Trigger $NoonDailyTrigger -ScheduledJobOption $JobOption
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Register-ScheduledJob -Name 'Update PowerShell Help' -ScriptBlock {>> Update-Help>>} -Trigger \$NoonDailyTrigger -ScheduledJobOption \$JobOption was run. The output shows a table with one row:

Id	Name	JobTriggers	Command	Enabled
1	Update Power...	1	...	True

Figure 16-33

This job can be viewed in task scheduler under **Task Scheduler (local) > Task Scheduler Library > Microsoft > Windows > PowerShell > ScheduledJobs**.

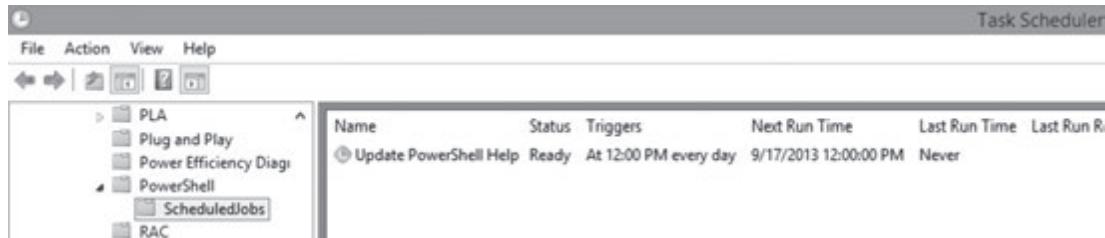
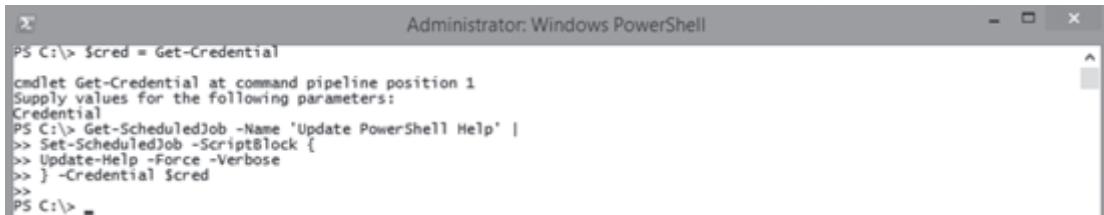


Figure 16-34

We'll make some modifications to our scheduled job because we've decided to change the command that it runs. We're going to specify the **-Force** parameter in case we want to update our Help more than once a day, and the **-Verbose** parameter so that detailed output about what was updated will be available in the results. We're also going to specify credentials so that the job can run regardless of whether or not we're logged in when the job is scheduled to start.

```
$cred = Get-Credential  
  
Get-ScheduledJob -Name 'Update PowerShell Help' |  
Set-ScheduledJob -ScriptBlock {  
    Update-Help -Force -Verbose  
} -Credential $cred
```

## Background and scheduled jobs

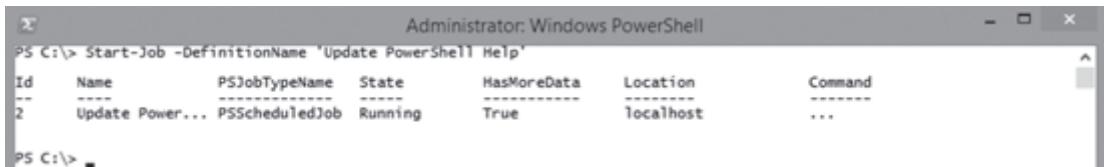


```
Administrator: Windows PowerShell
PS C:\> $cred = Get-Credential
cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
PS C:\> Get-ScheduledJob -Name 'Update PowerShell Help' |
>> Set-ScheduledJob -ScriptBlock {
>> Update-Help -Force -Verbose
>> } -Credential $cred
>>
PS C:\>
```

Figure 16-35

We'll go ahead and manually start the "Update PowerShell Help" scheduled job, by using the **Start-Job** cmdlet.

```
Start-Job -DefinitionName 'Update PowerShell Help'
```

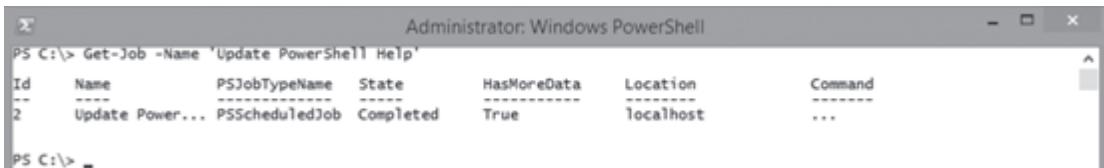


```
Administrator: Windows PowerShell
PS C:\> Start-Job -DefinitionName 'Update PowerShell Help'
Id     Name          PSJobTypeName   State      HasMoreData  Location        Command
--     --          PSScheduledJob  Running    True         Localhost      ...
2     Update Power... PSScheduledJob  Running    True         Localhost      ...
PS C:\>
```

Figure 16-36

By using the **Get-Job** cmdlet to check the job status, we can see that our "Update PowerShell Help" job is complete.

```
Get-Job -Name 'Update PowerShell Help'
```



```
Administrator: Windows PowerShell
PS C:\> Get-Job -Name 'Update PowerShell Help'
Id     Name          PSJobTypeName   State      HasMoreData  Location        Command
--     --          PSScheduledJob  Completed  True         Localhost      ...
2     Update Power... PSScheduledJob  Completed  True         Localhost      ...
PS C:\>
```

Figure 16-37

We'll now retrieve the results, by using the **Receive-Job** cmdlet.

```
Get-Job -Name 'Update PowerShell Help' | Receive-Job
```

```

PS C:\> Get-Job -Name 'Update PowerShell Help' | Receive-Job
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?LinkId=286813"
VERBOSE: Your connection has been redirected to the following URI:
"http://download.microsoft.com/download/1/F/9/1F9A2C13-4099-4BF3-9FD4-A56550883298/"
VERBOSE: AppBackgroundTask: Updated C:\Windows\system32\WindowsPowerShell\v1.0\Modules\AppBackgroundTask\en-US\microsoft.windows.appbackgroundtask.commands.dll-help.xml. Culture en-US Version 4.0.3.0
VERBOSE: AppBackgroundTask: Updated C:\Windows\system32\WindowsPowerShell\v1.0\Modules\AppBackgroundTask\en-US\PS_BackgroundTask.cdxml-help.xml. Culture en-US Version 4.0.3.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?LinkId=285540"
VERBOSE: Your connection has been redirected to the following URI:
"http://download.microsoft.com/download/B/1/9/B1964F35-3C65-4E55-85F4-A085B0724F12/"
VERBOSE: AppLocker: Updated C:\Windows\system32\WindowsPowerShell\v1.0\Modules\AppLocker\en-US\Microsoft.Security.ApplicationId.PolicyManagement.Cmdlets.dll-help.xml. Culture en-US Version 4.0.2.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?LinkId=285736"
VERBOSE: Your connection has been redirected to the following URI:
"http://download.microsoft.com/download/0/E/C/0ECE0473-E58E-4FFC-B186-D37378BC4414/"
VERBOSE: Appx: Updated C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Appx\en-US\Microsoft.Windows.Appx.PackageManager.Commands.dll-help.xml. Culture en-US Version 4.0.2.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?LinkId=285737"
VERBOSE: Your connection has been redirected to the following URI:
"http://download.microsoft.com/download/3/5/9/35975FAF-0537-41AB-8E26-A04E9109E49D/"
VERBOSE: BestPractices: Updated C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BestPractices\en-US\Microsoft.BestPractices.Cmdlets.dll-help.xml. Culture en-US Version 4.0.2.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?LinkId=285738"
VERBOSE: Your connection has been redirected to the following URI:
"http://download.microsoft.com/download/5/C/8/5C86AED1-6DE3-4B7C-8588-3C2A0A83991C/"
VERBOSE: BitLocker: Updated C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BitLocker\en-US\BitLocker.psml-help.xml. Culture en-US Version 4.0.3.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?LinkId=285541"
VERBOSE: Your connection has been redirected to the following URI:
"http://download.microsoft.com/download/D/2/F/D2F99097-C196-43D8-AFC6-D6417109AFCC/"

```

Figure 16-38

There are cmdlets for disabling scheduled jobs (**Disable-ScheduledJob**) and unregistering scheduled jobs (**Unregister-ScheduledJob**).

By placing these commands inside of **Invoke-Command**, it is also possible to set up and manage scheduled jobs on remote servers—so the possibilities are endless.

## Exercise 16 — PowerShell jobs

You've had several different responsibilities as of late and you've written some great PowerShell scripts to automate much of those responsibilities, but you're finding that many of the scripts you're running are tying up your PowerShell console. As a result, you're unable to do any other work, without resorting to opening another PowerShell console.

You've recently read about the Jobs feature in PowerShell and you want to try it out in a controlled environment to see if it might better meet your needs, instead of having multiple PowerShell consoles open.

You also have a couple of scripts that you've written and that you'd like to setup as scheduled jobs.

### Task 1

Create a local background job to retrieve a list of all of the hotfixes that are installed on the local computer.

### Task 2

Check to see if the job in the previous task has completed.

### Task 3

Receive the results of the job that was created in task 1, without clearing the results, so that they can be retrieved again.

### Task 4

Store the results from the job that was created in task 1 in a variable, and then display the contents of the variable in the PowerShell console.

### Task 5

Remove the job that was created in task 1.

### Task 6

Repeat task 1 through 5, creating remote jobs to one or more remote computers in your environment. If you only have one computer in your test environment, specify **\$Env:ComputerName** as the remote computer name.

### Task 7

Verify the **bits** service is running. Create a scheduled job that will stop the **bits** service one time, five minutes from now. Wait five minutes to verify the scheduled job completed successfully or start the scheduled job manually.

### Task 8

Start the **bits** service that was stopped in task 7. Remove the scheduled job that was created in task 7.



## Chapter 17

# **Making tools: Turning a script into a reusable tool**

## **Making a better script with parameters**

What is a parameter? In PowerShell, a parameter allows users to provide input or select options for scripts, functions, or cmdlets.

Careful consideration should be taken when writing PowerShell scripts, so that their reusability is maximized. While writing scripts for reusability may initially take more time, it will minimize the number of times that you have to modify a script and it will also reduce the number of scripts that you have to maintain. The following example returns a list of local user accounts from a server named SQL01.

```
Get-CimInstance -ComputerName sql01 -ClassName Win32_Account -Filter "SIDType=1 and LocalAccount=$true"
```

This script has been saved as **Get-LocalUser.ps1** in the C:\Scripts folder on your local computer. As you can see in the following example, the script works great.

```
.\Get-LocalUser.ps1
```

```

Administrator: Windows PowerShell
PS C:\Scripts> .\Get-LocalUser.ps1
Name        Caption      AccountType      SID          Domain       PSComputerName
----        -----      -----          ---          -----       -----
Administrator SQL01\Administrator 512   S-1-5-21-34405914... SQL01      sql01
Guest       SQL01\Guest    512   S-1-5-21-34405914... SQL01      sql01

```

Figure 17-1

At least, it works great until you need to retrieve the local user accounts from a server with a different name. You could write another script and hard code the name of the other server into it and save it as a new script with a different file name. However, then you would have two copies of almost the same script saved as two different files, which means you would be doubling the amount of code you'll have to maintain if something ever needs to be changed. You could also just type this command into the console and run it, but as your scripts become longer and more advanced, neither of these options are desirable.

What we need to do is define the computer name as a parameter for the script. We've made some simple changes to our script in the following example. A parameter block has been added, along with a **ComputerName** variable inside that block. Then the hardcoded computer name of SQL01 has been replaced with the **ComputerName** variable.

```

param (
    $ComputerName
)

Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter "SIDType=1 and
LocalAccount=$true"

```

We now have a parameterized script. In the PowerShell console, we can type the name of the script followed by a space, then a dash, and then pressing the **Tab** key automatically puts the **-ComputerName** parameter on the line, which we'll follow with a value for the **-ComputerName** parameter (web01).

```
.\Get-LocalUser.ps1 -ComputerName web01
```

```

Administrator: Windows PowerShell
PS C:\Scripts> .\Get-LocalUser.ps1 -ComputerName web01
Name        Caption      AccountType      SID          Domain       PSComputerName
----        -----      -----          ---          -----       -----
Administrator WEB01\Administrator 512   S-1-5-21-29893675... WEB01      web01
Guest       WEB01\Guest    512   S-1-5-21-29893675... WEB01      web01

```

Figure 17-2

This parameter works just like the parameters for the built-in cmdlets. Speaking of the built-in cmdlets, when creating parameters for your scripts, see what parameter names the built-in cmdlets use before choosing a parameter name. This will help keep your scripts consistent with the rest of PowerShell and ultimately make them easier to use, especially if you're sharing them with others.

There are other **SIDTypes** as well, so we'll parameterize that value just in case we ever want to return a different type of account. We've also specified a **datatype** for each of our parameters. The extra set of square brackets that we specified after the string **datatype** for the **-ComputerName** parameter means that it can accept multiple strings or an array of strings. We specified a **datatype** of **integer** for the **-SIDType** parameter.

```
param (
    [string[]]$ComputerName,
    [int]$SIDType = 1
)

Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter "SIDType=$SIDType and LocalAccount=$true"
```

Caption	Domain	Name	SID	PSComputerName
WEB01\BUILTIN	WEB01	BUILTIN	S-1-5-32	web01
SQL01\BUILTIN	SQL01	BUILTIN	S-1-5-32	sql01

Figure 17-3

Specifying a **datatype** for the parameters in your scripts is a best practice because it's a form of input validation for the values that are specified for the parameters. For example, if a user enters a non-numeric value for the **-SIDType** parameter, the script will terminate at that point and return an error to the user, instead of continuing and attempting to retrieve the local users where the **-SIDType** is located (for example, "one" for each of the servers specified via the **-ComputerName** parameter).

```
.\Get-LocalUser.ps1 -ComputerName web01, sql01 -SIDType one
```

Figure 17-4

## Turning a script into a function

At this point, let's go ahead and turn our `Get-LocalUser.ps1` script into a function.

```
function Get-LocalUser {
    param (
        [string[]]$ComputerName,
        [int]$SIDType = 1
    )
    Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter "SIDType=$SIDType and LocalAccount=$true"
}
```

That was easy, right? We simply specified the function keyword, followed by a name for the function, and enclosed our previous script in curly braces. Functions should be named using singular Verb-Noun syntax, just like cmdlets. The verb should be an approved verb. A list of approved verbs can be retrieved by running **Get-Verb | Where-Object group -eq common** from within PowerShell.

We saved our **Get-LocalUser** function as Get-LocalUser.ps1, replacing our existing script. We run the script and nothing happens. Why?

```
.\Get-LocalUser.ps1
```



Figure 17-5

Nothing happens because the function loads into memory in the script scope, then the script ends and exits, taking the script scope with it, and then the function is gone.

There are a couple of ways to test functions. The first way is to call the function at the end of the script, so that when the script loads the function into the script scope, it executes the function before the script ends and exits.

```
function Get-LocalUser {
    param (
        [string[]]$ComputerName,
        [int]$SIDType = 1
    )
    Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter "SIDType=$SIDType and LocalAccount=$true"
}
Get-LocalUser
```

Name	Caption	AccountType	SID	Domain
Administrator	PC02\Administrator	512	S-1-5-21-3940912009-34...	PC02
Guest	PC02\Guest	512	S-1-5-21-3940912009-34...	PC02
Mike	PC02\Mike	512	S-1-5-21-3940912009-34...	PC02

Figure 17-6

The problem with calling the function at the end of the script is that we're back to having to manually modify the script each time we want to query a different server or specify a different **SIDType**. It's also a common practice to place multiple functions into a single script, which presents another problem for this solution.

The second method for testing a function is to dot source the script, which loads the function into the global scope. The function can then be used for the life of the PowerShell session. At this point, we've removed the line at the bottom of the script where we called the function. Assuming that you're in the same directory as the script, a script is dot sourced by running the script with an extra dot, and then a space before the normal dot slash that specifies the current folder, followed by the script name and extension. Note that the space between the two dots is required, as shown in Figure 17-7.

```
. .\Get-LocalUser.ps1
```

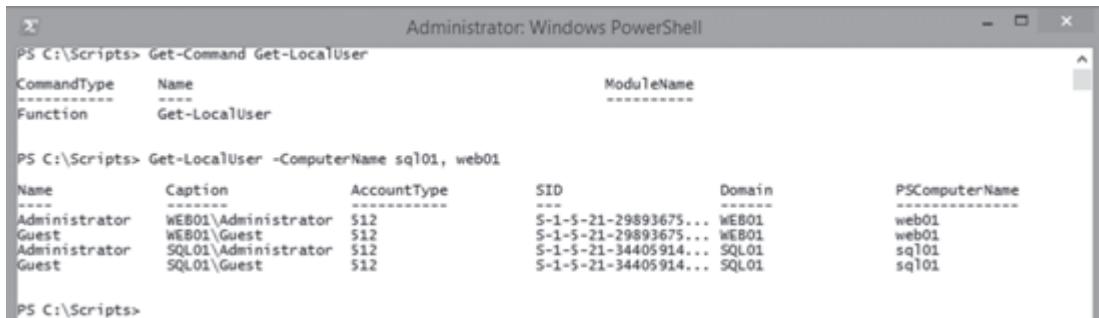
Figure 17-7

It appears that the script didn't do anything again, but now we can see the **Get-LocalUser** function exists when we look for it using the **Get-Command** cmdlet. We can also call the function that retrieves the local user accounts for the specified servers.

```
Get-Command Get-LocalUser

Get-LocalUser -ComputerName sql01, web01
```

## Windows PowerShell: TFM



```
Administrator: Windows PowerShell
PS C:\Scripts> Get-Command Get-LocalUser
 CommandType      Name          ModuleName
 Function        Get-LocalUser

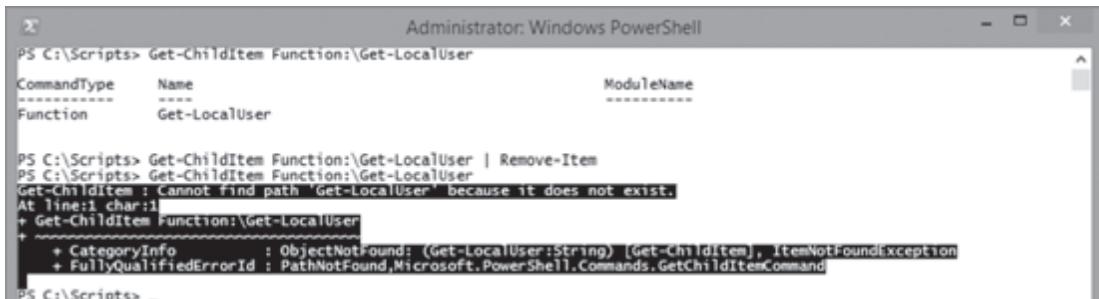
PS C:\Scripts> Get-LocalUser -ComputerName sql01, web01
Name      Caption      AccountType      SID          Domain      PSComputerName
----      -----      -----          ---          ----          -----
Administrator  WEB01\Administrator  512          S-1-5-21-29893675...  WEB01      web01
Guest       WEB01\Guest        512          S-1-5-21-29893675...  WEB01      web01
Administrator  SQL01\Administrator  512          S-1-5-21-34405914...  SQL01      sql01
Guest       SQL01\Guest        512          S-1-5-21-34405914...  SQL01      sql01

PS C:\Scripts>
```

Figure 17-8

One of the problems with dot sourcing a script to load its functions into memory is that the only way to remove the functions, short of closing PowerShell, is to remove them from the function **psdrive**.

```
Get-ChildItem Function:\Get-LocalUser
Get-ChildItem Function:\Get-LocalUser | Remove-Item
Get-ChildItem Function:\Get-LocalUser
```



```
Administrator: Windows PowerShell
PS C:\Scripts> Get-ChildItem Function:\Get-LocalUser
 CommandType      Name          ModuleName
 Function        Get-LocalUser

PS C:\Scripts> Get-ChildItem Function:\Get-LocalUser | Remove-Item
PS C:\Scripts> Get-ChildItem Function:\Get-LocalUser
Get-ChildItem : Cannot find path "Get-LocalUser" because it does not exist.
At line:1 char:1
+ Get-ChildItem Function:\Get-LocalUser
+ CategoryInfo          : ObjectNotFound: (Get-LocalUser:String) [Get-ChildItem], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
PS C:\Scripts>
```

Figure 17-9

We'll add a second function, named **Get-TimeZone**, to our script and rename the script to something more generic. Since the script now contains more than one function or tool, it's becoming a toolkit. The C:\Scripts\ServerToolkit.ps1 script now contains the following functions.

```
function Get-LocalUser {
    param (
        [string[]]$ComputerName,
        [int]$SIDType = 1
    )
    Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter "SIDType=$SIDType and LocalAccount=$true"
}
```

```
function Get-TimeZone {
    param (
        [string[]]$ComputerName
    )

    Get-CimInstance -ClassName Win32_TimeZone -ComputerName $ComputerName |
        Select-Object -Property @{label='TimeZone';expression={$_._Caption}}, pscomputername
}
```

We just sourced the ServerToolkit.ps1 script, and then we ran both the **Get-TimeZone** and **Get-LocalUser** functions.

```
. .\ServerToolkit.ps1
Get-TimeZone -ComputerName sql01, web01
Get-LocalUser -ComputerName sql01, web01
```

```
Administrator: Windows PowerShell
PS C:\Scripts> . .\ServerToolkit.ps1
PS C:\Scripts> Get-TimeZone -ComputerName sql01, web01
TimeZone                                PSComputerName
-----                                -----
(UTC-06:00) Central Time (US & Canada)    sql01
(UTC-06:00) Central Time (US & Canada)    web01

PS C:\Scripts> Get-LocalUser -ComputerName sql01, web01
Name          Caption      AccountType      SID           Domain      PSComputerName
----          -----      -----          --           --           --
Administrator  WEB01\Administrator  512          S-1-5-21-29893675...  WEB01      web01
Guest         WEB01\Guest       512          S-1-5-21-29893675...  WEB01      web01
Administrator  SQL01\Administrator  512          S-1-5-21-34405914...  SQL01     sq101
Guest         SQL01\Guest       512          S-1-5-21-34405914...  SQL01     sq101

PS C:\Scripts> _
```

Figure 17-10

## Variables and scope

As we previously discussed, a parameter is nothing more than a variable defined inside of the **param** block.

## Advanced functions: Turning a function into an advanced function

To turn a function into an advanced function, add the **CmdletBinding** attribute just before the function's parameter block.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [string[]]$ComputerName,
        [int]$SIDType = 1
    )
    Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter "SIDType=$SIDType and LocalAccount=$true"
}
```

The **CmdletBinding** attribute enables several options for functions, which makes them seem more like one of the built-in cmdlets.

If you take a look at the available parameters of our **Get-LocalUser** function, prior to adding the **CmdletBinding** attribute, you'll see that there are only two parameters available, which are the two that we defined.

```
Get-Command Get-LocalUser | Select-Object -ExpandProperty parameters
```

Key	Value
ComputerName	System.Management.Automation.ParameterMetadata
SIDType	System.Management.Automation.ParameterMetadata

Figure 17-11

After adding the **CmdletBinding** attribute and dot sourcing our ServerTools.ps1 script again, you can see that there are many more parameters available. That's something you'll want to remember—each time you make a modification to the function in your script, you'll need to dot source the script again, to load the new version of the function into memory, replacing the old version that was previously in memory; otherwise your changes won't show up when trying to use the function.

These additional parameters, that are now available after adding the **CmdletBinding** attribute, are called common parameters.

```
Get-Command Get-LocalUser | Select-Object -ExpandProperty parameters
```

Key	Value
ComputerName	System.Management.Automation.ParameterMetadata
SIDType	System.Management.Automation.ParameterMetadata
Verbose	System.Management.Automation.ParameterMetadata
Debug	System.Management.Automation.ParameterMetadata
ErrorAction	System.Management.Automation.ParameterMetadata
WarningAction	System.Management.Automation.ParameterMetadata
ErrorVariable	System.Management.Automation.ParameterMetadata
WarningVariable	System.Management.Automation.ParameterMetadata
OutVariable	System.Management.Automation.ParameterMetadata
OutBuffer	System.Management.Automation.ParameterMetadata
PipelineVariable	System.Management.Automation.ParameterMetadata

Figure 17-12

The **PipelineVariable** parameter shown in Figure 17-12 is a new common parameter, which was added in PowerShell version 4. For more information about this new common parameter, as well as the other common parameters, see the `about_CommonParameters` Help topic in PowerShell.

Turning a function into an advanced function by adding the **CmdletBinding** attribute also enables the use of additional attributes in your functions, many of which we'll be covering in the next section.

## Adding parameter attributes

### Mandatory

We're going to make the **-ComputerName** parameter of our **Get-LocalUser** function mandatory, by adding the **mandatory** parameter attribute to that particular parameter.

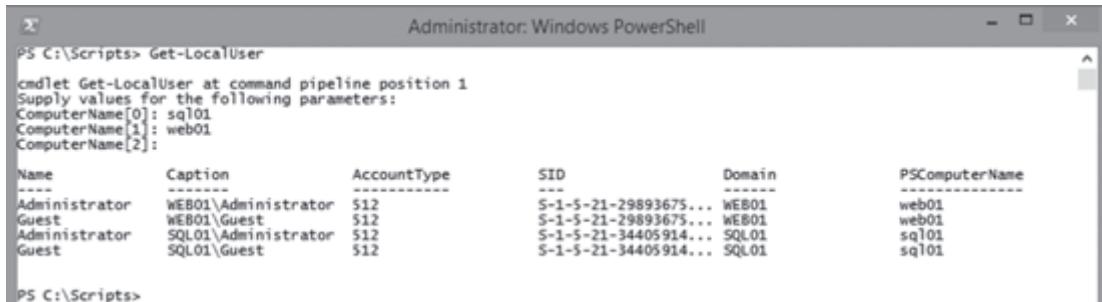
```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory)]
        [string[]]$ComputerName,
        [int]$SIDType = 1
    )

    Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter "SIDType=$SIDType and LocalAccount=$true"
}
```

Prior to PowerShell version 3, the syntax for the line to make the parameter mandatory would have been: `[parameter(Mandatory=$true)]`. However, beginning with PowerShell version 3, it works almost like a **-Switch** parameter, where specifying **mandatory** makes it true by default. So, there is no need to specify `=$true`, unless backwards compatibility with previous versions of PowerShell is desired.

Now, when we attempt to run our **Get-LocalUser** function without specifying a value for the **-ComputerName** parameter, we're prompted to enter a computer name. Since we defined the **ComputerName** with a **datatype** of one or more strings, we'll be prompted for additional computer names until we press **Enter** on an empty line.

```
Get-LocalUser
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\Scripts> Get-LocalUser is run. A cmdlet help message is displayed, asking for values for ComputerName[0], ComputerName[1], and ComputerName[2]. The command then lists local users across three computers: web01, web01, and sql01. The output is as follows:

Name	Caption	AccountType	SID	Domain	PSComputerName
Administrator	WEB01\Administrator	512	S-1-5-21-29893675...	WEB01	web01
Guest	WEB01\Guest	512	S-1-5-21-29893675...	WEB01	web01
Administrator	SQL01\Administrator	512	S-1-5-21-34405914...	SQL01	sql01
Guest	SQL01\Guest	512	S-1-5-21-34405914...	SQL01	sql01

Figure 17-13

One thing to note is that the **mandatory** parameter and default values are mutually exclusive. You can specify a default value, as we did with our **SIDType** parameter, but the default value will be ignored and the user of the function will be prompted if they don't provide a value when calling the function, even though a default value was set in the script.

## ValueFromPipeline

We want to accept input for the **-ComputerName** parameter via pipeline input so that we can retrieve the computer names from a text file or from Active Directory and pipe the output of the command that retrieves those names into our function. In addition to adding the **-ValueFromPipeline** parameter attribute, we've also added a **Process** block to our function, as shown in Figure 17-14. The **Process** block will run one time for each object in the pipeline.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [int]$SIDType = 1
    )
    PROCESS {
        Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter
        "SIDType=$SIDType and LocalAccount=$true"
    }
}
```

```

Administrator: Windows PowerShell
PS C:\Scripts> Get-ADComputer -Filter "OperatingSystem -like 'Windows Server 2012 R2*' |
>> Select-Object -ExpandProperty DNSHostName |
>> Get-LocalUser
>>

```

Name	Caption	AccountType	SID	Domain	PSComputerName
Administrator	SQL01\Administrator	512	S-1-5-21-34405914...	SQL01	sql01.mikefrobbi...
Guest	SQL01\Guest	512	S-1-5-21-34405914...	SQL01	sql01.mikefrobbi...
Administrator	WEB01\Administrator	512	S-1-5-21-29893675...	WEB01	web01.mikefrobbi...
Guest	WEB01\Guest	512	S-1-5-21-29893675...	WEB01	web01.mikefrobbi...

```

PS C:\Scripts>

```

Figure 17-14

## ValidateNotNullOrEmpty

We want to make sure that a value is provided for our **-SIDType** parameter. If this parameter was defined as a string, we would use **ValidateNotNullOrEmpty** to accomplish the task, as shown in Figure 17-15, but both an empty string "" or a null value **\$null** will be implicitly converted to an integer and equal 0. So this type of parameter validation doesn't work as expected when the **datatype** is an integer.

```

function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [ValidateNotNullorEmpty()]
        [int]$SIDType = 1
    )
    PROCESS {
        Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter
        "$SIDType=$SIDType and LocalAccount=$true"
    }
}

```

```

Administrator: Windows PowerShell
PS C:\Scripts> Get-LocalUser -ComputerName sql01 -SIDType ''
PS C:\Scripts>

```

Figure 17-15

As you can see in the results of Figure 17-15, the parameter validation wasn't tripped, even though we provided an empty string for the value of **SIDType**.

If you wanted to prevent null values, you can define **SIDType** as a nullable integer.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [ValidateNotNullOrEmpty()]
        [Nullable[int]]$SIDType = 1
    )
    PROCESS {
        Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter
        "SIDType=$SIDType and LocalAccount=$true"
    }
}
```

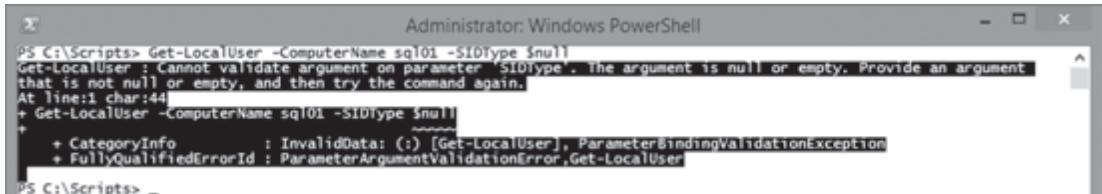


Figure 17-16

This prevents a null value from being specified, but an empty string would still be allowed. As previously stated, if **-SIDType** was a string, the **-ValidateNotNullOrEmpty** parameter attribute would have met our needs. Since this is an attribute that you'll commonly use, we want to make sure that you are aware of these issues when you're working with integers.

## ValidateRange

Since our **-SIDType** parameter is an integer and after taking a look at the Win32\_UserAccount class on MSDN: <http://tinyurl.com/q6hkquiv>, the only valid values for this particular property are 1 through 9, we are going to use the **-ValidateRange** parameter attribute to restrict the numeric values that are allowed, as shown in Figure 17-17. We've also removed the **-ValidateNotNullOrEmpty** parameter attribute and changed the **-SIDType** datatype back to integer.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [ValidateRange(1,9)]
        [int]$SIDType = 1
    )
    PROCESS {
        Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter
        "SIDType=$SIDType and LocalAccount=$true"
    }
}
```

```
Administrator: Windows PowerShell
PS C:\Scripts> Get-LocalUser -ComputerName sql01 -SIDType 10
Get-LocalUser : Cannot validate argument on parameter 'SIDType'. The 10 argument is greater than the maximum allowed range of 9. Supply an argument that is less than or equal to 9 and then try the command again.
At line:1 char:44
+ Get-LocalUser -ComputerName sql01 -SIDType 10
+ ~~~~~
+ CategoryInfo          : InvalidData: (:) [Get-LocalUser], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,Get-LocalUser
PS C:\Scripts>
```

Figure 17-17

As you can see, a value of 10 is not accepted, since the value range we specified was 1 through 9. This will also prevent empty or null values from being accepted since, as we previously stated, they're implicitly converted to 0. The error message shown in Figure 17-18, where we attempt to specify an empty value for **-SIDType**, confirms that an empty string is indeed implicitly converted to 0.

```
Get-LocalUser -ComputerName sql01 -SIDType ''
```

```
Administrator: Windows PowerShell
PS C:\Scripts> Get-LocalUser -ComputerName sql01 -SIDType ''
Get-LocalUser : Cannot validate argument on parameter 'SIDType'. The '' argument is less than the minimum allowed range of 1. Supply an argument that is greater than or equal to 1 and then try the command again.
At line:1 char:44
+ Get-LocalUser -ComputerName sql01 -SIDType ''
+ ~~~~~
+ CategoryInfo          : InvalidData: (:) [Get-LocalUser], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,Get-LocalUser
PS C:\Scripts>
```

Figure 17-18

## DontShow

New to PowerShell version 4 is the **-DontShow** parameter attribute, which prevents a parameter from showing up in tabbed expansion or IntelliSense. This is the perfect attribute to use for our **-SIDType** parameter, since we don't really want the users of our tool to change the **-SIDType** parameter value from the default value that we've specified, unless they've been specifically instructed to do so.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1,9)]
        [int]$SIDType = 1
    )
    PROCESS {
        Get-CimInstance -ComputerName $ComputerName -ClassName Win32_Account -Filter
        "SIDType=$SIDType and LocalAccount=$true"
    }
}
```

While this prevents the **-SIDType** parameter from showing up via tabbed expansion or IntelliSense, it doesn't prevent it from showing up in the Help or when using the **Get-Command** cmdlet to view the details of the function.

## Scripts versus modules

In our opinion, scripts provide an easy mechanism to test functions, by using the dot source method that we discussed earlier in this chapter. However, once you're ready to go live and start using your scripts in production environments, you should really combine the functions into a module. Creating a module also makes sharing your functions easier.

## Turning your functions into a module

Now we'll show you how to turn the C:\Scripts\ServerToolkit.ps1 script that we've been using throughout this chapter into a script module. This is the script that we currently have our **Get-LocalUser** and **Get-TimeZone** functions saved in.

In Figure 17-19, we've simply opened the script in SAPIEN PowerShell Studio 2014, selected **Module Script Files (\*.psm1)** for the **Save as type**, and then saved the file in the **Documents\WindowsPowerShell\Modules\ServerToolkit** folder. The name of the folder that you create in the **Modules** folder must be identical to the base name of the module in order for PowerShell to be able to automatically locate it.

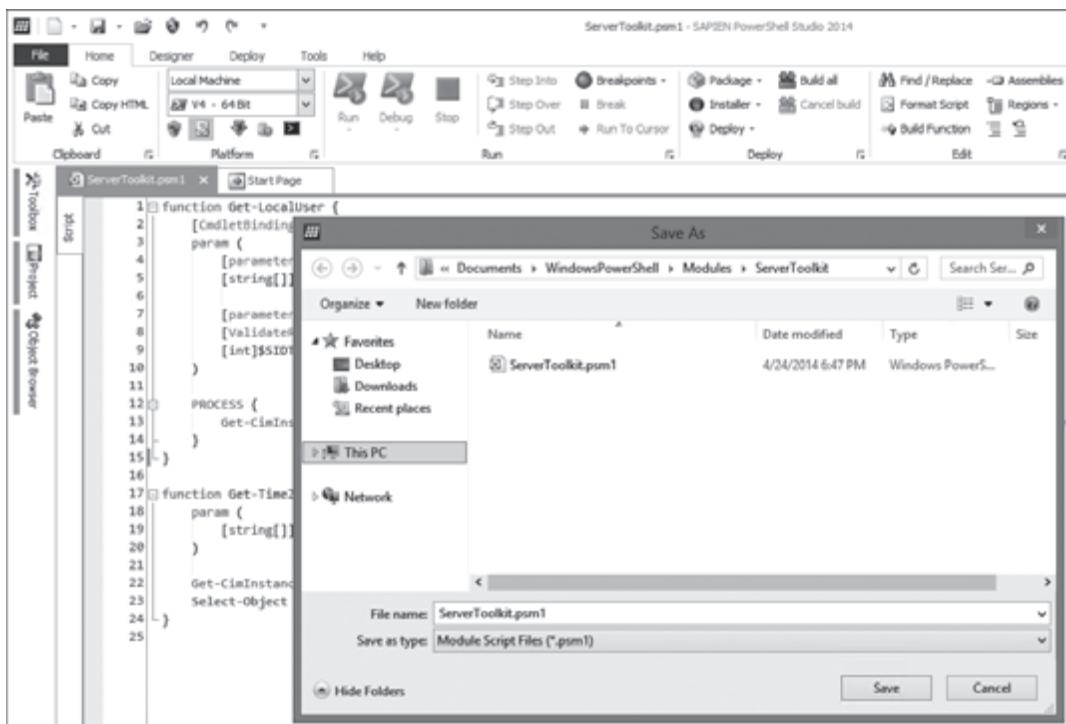
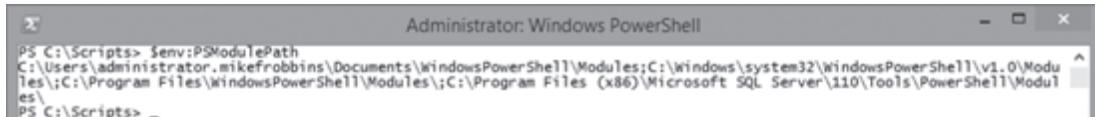


Figure 17-19

## Making tools: Turning a script into a reusable tool

Saving the module to this location is very important because it's one of the two default locations that PowerShell looks for modules. The **\$env:PSModulePath** variable contains the locations shown in Figure 17-20.

```
$env:PSModulePath
```



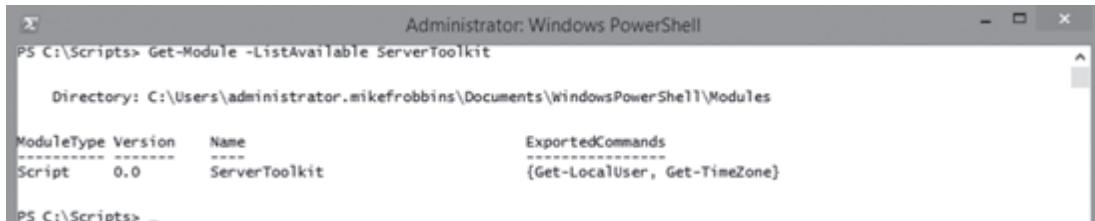
```
Administrator: Windows PowerShell
PS C:\Scripts> $env:PSModulePath
C:\Users\administrator.mikefrobbins\Documents\WindowsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules\;C:\Program Files\WindowsPowerShell\Modules\;C:\Program Files (x86)\Microsoft SQL Server\110\Tools\PowerShell\Modules\

PS C:\Scripts>
```

Figure 17-20

We can now see that our module is available and that it contains two exported commands.

```
Get-Module -ListAvailable ServerToolkit
```



```
Administrator: Windows PowerShell
PS C:\Scripts> Get-Module -ListAvailable ServerToolkit

Directory: C:\Users\administrator.mikefrobbins\Documents\WindowsPowerShell\Modules

ModuleType Version Name                                ExportedCommands
-----  -----  ----                                -----
Script     0.0   ServerToolkit                      {Get-LocalUser, Get-TimeZone}

PS C:\Scripts>
```

Figure 17-21

## Adding a manifest for your module

We recommend, and consider it to a best practice, to create a module manifest file for each module that you create. A module manifest describes the contents and system requirements of the module. We'll use the **New-ModuleManifest** cmdlet to create our module manifest file.

```
New-ModuleManifest -Path .\ServerToolkit.psd1 -PowerShellVersion 4.0 -FunctionsToExport Get-LocalUser -RootModule ServerToolkit
```



```
Administrator: Windows PowerShell
PS C:\Scripts> New-ModuleManifest -Path .\ServerToolkit.psd1 -PowerShellVersion 4.0 -FunctionsToExport Get-LocalUser -RootModule ServerToolkit
PS C:\Scripts>
```

Figure 17-22

## Windows PowerShell: TFM

This file should have the same base name as the actual module and it should be saved in the same folder as the module.

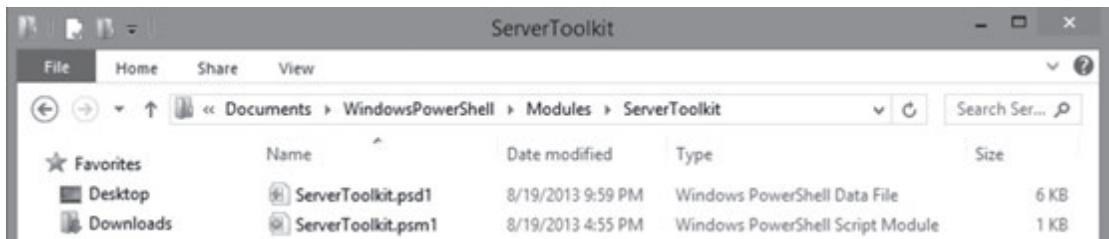


Figure 17-23

Once the manifest file is created, it can be opened with an editor, such as PowerShell Studio 2014 or PrimalScript 2014, and modified as needed. You can see there are several values that were set automatically.

```
#  
# Module manifest for module 'ServerToolkit'  
#  
# Generated by: Administrator  
#  
# Generated on: 8/19/2013  
#  
  
#{@  
  
# Script module or binary module file associated with this manifest.  
RootModule = 'ServerToolkit'  
  
# Version number of this module.  
ModuleVersion = '1.0'  
  
# ID used to uniquely identify this module  
GUID = '1c2977d1-8d3f-45a6-9c73-482c43e27eff'  
  
# Author of this module  
Author = 'Administrator'  
  
# Company or vendor of this module  
CompanyName = 'Unknown'  
  
# Copyright statement for this module  
Copyright = '(c) 2013 Administrator. All rights reserved.'  
  
# Description of the functionality provided by this module  
# Description = ''  
  
# Minimum version of the Windows PowerShell engine required by this module  
PowerShellVersion = '4.0'  
  
# Name of the Windows PowerShell host required by this module  
# PowerShellHostName = ''  
  
# Minimum version of the Windows PowerShell host required by this module
```

```
# PowerShellHostVersion = ''  
  
# Minimum version of Microsoft .NET Framework required by this module  
# DotNetFrameworkVersion = ''  
  
# Minimum version of the common language runtime (CLR) required by this module  
# CLRVersion = ''  
  
# Processor architecture (None, X86, Amd64) required by this module  
# ProcessorArchitecture = ''  
  
# Modules that must be imported into the global environment prior to importing this module  
# RequiredModules = @()  
  
# Assemblies that must be loaded prior to importing this module  
# RequiredAssemblies = @()  
  
# Script files (.ps1) that are run in the caller's environment prior to importing this module.  
# ScriptsToProcess = @()  
  
# Type files (.ps1xml) to be loaded when importing this module  
# TypesToProcess = @()  
  
# Format files (.ps1xml) to be loaded when importing this module  
# FormatsToProcess = @()  
  
# Modules to import as nested modules of the module specified in RootModule/ModuleToProcess  
# NestedModules = @()  
  
# Functions to export from this module  
FunctionsToExport = 'Get-LocalUser'  
  
# Cmdlets to export from this module  
CmdletsToExport = '*'  
  
# Variables to export from this module  
VariablesToExport = '*'  
  
# Aliases to export from this module  
AliasesToExport = '*'  
  
# List of all modules packaged with this module  
# ModuleList = @()  
  
# List of all files packaged with this module  
# FileList = @()  
  
# Private data to pass to the module specified in RootModule/ModuleToProcess  
# PrivateData = ''  
  
# HelpInfo URI of this module  
# HelpInfoURI = ''  
  
# Default prefix for commands exported from this module. Override the default prefix using Import-  
Module -Prefix.  
# DefaultCommandPrefix = ''  
  
}
```

## Windows PowerShell: TFM

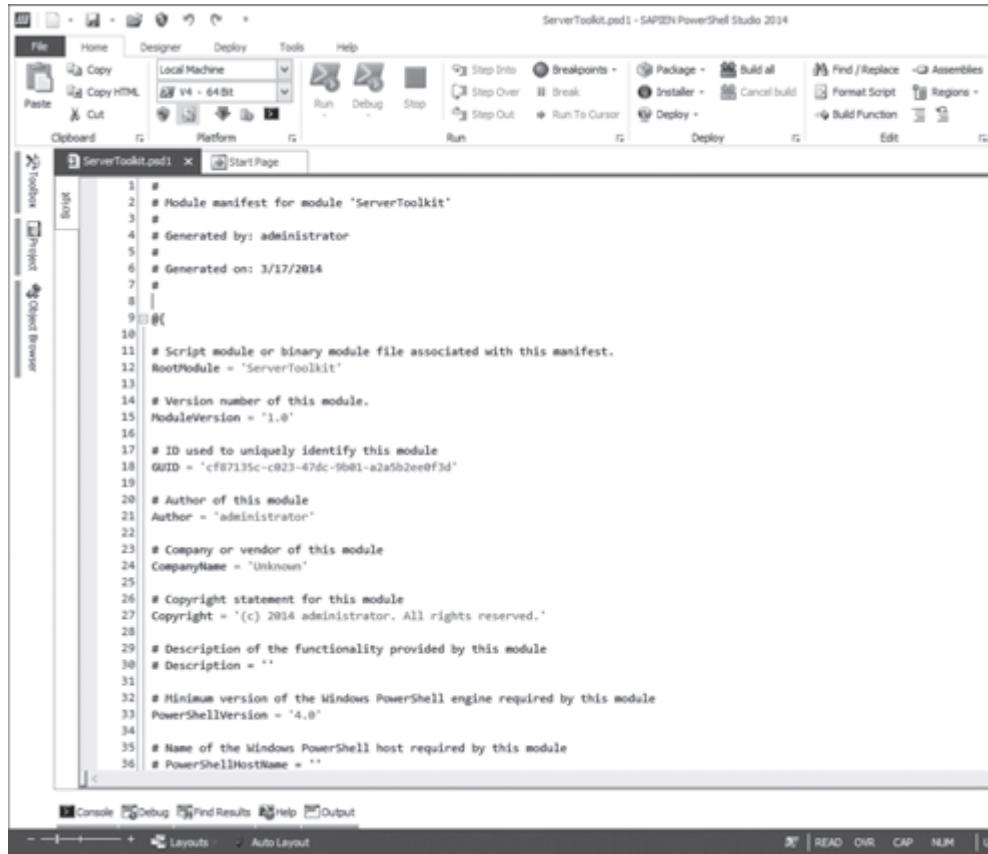


Figure 17-24

Remember when we told you to use approved verbs for your functions? Well, we're going to make two changes to our ServerToolkit module, the first to the manifest file, to have it export all commands, and the second to our **Get-TimeZone** function, renaming it to **Aquire-TimeZone**. We're doing this to prove a point because "Aquire" is not an approved verb. Now, we'll reimport our module using the **-Force** parameter, since it was already loaded.

```
Import-Module -Name ServerToolkit -Force
```

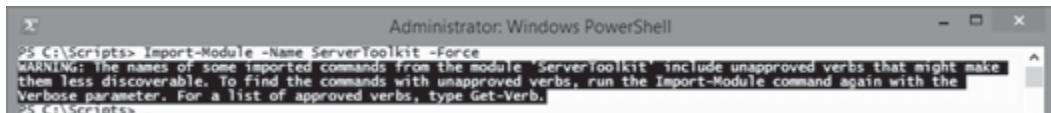


Figure 17-25

We hate to say we told you so, but we told you earlier to use approved verbs for your functions. You won't necessarily see any problems until you add them to a module; then in addition to us saying "we told you so," PowerShell will also say it's telling you so. So don't use unapproved verbs in your functions!

## Exercise 17 – Create a reusable tool

You've been tasked with taking a PowerShell one liner that you commonly use and turn it into a reusable tool that can be turned over to junior-level system admins. The one liner that you commonly use is shown below.

```
Get-CimInstance -ClassName Win32_BIOS | Select-Object -Property SerialNumber
```

This one liner returns the serial number of the local computer. The junior-level admins need to be able to retrieve the serial number of remote computers so that they can look up system information, as well as warranty information on the manufacturer's website.

### Task 1

Create a parameterized script that accepts the computer name as a parameter. Make the computer name default to the local computer.

### Task 2

Turn the script from Task 1 into a function.

### Task 3

Create a new module and add the function created in Task 2 to this new module.

### Task 4

Create a module manifest file for the module created in Task 3.



## Chapter 18

# Making tools: Creating object output

## Taking your cmdlets to the next level

The scripts and functions that you create look and feel like real cmdlets, but what happens when you run multiple commands that produce different types of output within the same script or function? To make a long story short, it results in a pipeline—which outputs chaos. What you need is for the commands within your script or function to be combined into a single custom object, so that the results are consistent.

## Output designed for the pipeline

Your functions should create one type of output and only one type of output. Let's imagine that your employer will be implementing some new software and they need you to perform a system inventory of all of their servers. The specific information that you need to inventory includes: the speed of the processor, number of logical CPU cores, amount of RAM, total disk space along with the amount of available free disk space from all of the logical hard drives, and what operating system the servers are running.

While it is possible to find all of the necessary WMI classes that contain this information by using PowerShell alone, products such as SAPIEN WMI Explorer make the task of locating the WMI classes that contain this information significantly easier, especially if you're new to WMI and/or PowerShell.

Using WMI Explorer, we can see that **-NumberofLogicalProcessors**, and **-TotalPhysicalMemory** are properties of the **Win32\_ComputerSystem** WMI class.

## Windows PowerShell: TFM

The screenshot shows the Windows PowerShell interface for WMI Explorer 2014. The title bar reads "pc02 - WMI Explorer 2014". The left sidebar lists various WMI namespaces and classes. The main pane displays the "Win32\_ComputerSystem" class with its properties.

**Class:**

- Win32\_ComputerSystem
- Win32\_ComputerSystemEvent
- Win32\_ComputerSystemProduct

**Description:**

The Win32\_ComputerSystem class represents a computer system operating in a Win32 environment. This event class represents events related to a computer system. The Win32\_ComputerSystemProduct class represents a product. This includes software and hardware used on this computer system.

**Property / Method:**

Property	CIMType / .NET Type	Description
NumberOfLogicalProcessors	uint32 / System.UInt32	The NumberOfLogicalProcessors property indicates the number of logical processors.
NumberOfProcessors	uint32 / System.UInt32	The NumberOfProcessors property indicates the number of physical processors.
OEMLogoBitmap	uint8 / System.Byte	The OEMLogoBitmap array holds the data for a bitmap created by the OEM.

**Output:**

Property	Value
NumberOfLogicalProcessors	4
NumberOfProcessors	1
OEMLogoBitmap	
PartOfDomain	True
PauseAfterReset	3932100000
PCSystemType	1
PowerManagementCapabilities	
PowerManagementSupported	
PowerOnPasswordStatus	3
PowerState	0
PowerSupplyState	3
PrimaryOwnerContact	
PrimaryOwnerName	Mike
ResetCapability	1
ResetCount	-1
ResetLimit	-1
Status	OK
SupportContactDescription	
SystemStartupDelay	
SystemStartupOptions	
SystemStartupSetting	
SystemType	x64-based PC
ThermalState	1
TotalPhysicalMemory	3220758528

Help Search results Query results

Figure 18-1

We've also located the disk information that we will need to fulfill that portion of the requirements. We located this information in the **Win32\_LogicalDisk** WMI class. One of the really useful features of SAPIEN WMI Explorer is that it allows you to create custom WMI queries and run them in PowerShell from WMI Explorer.

## Making tools: Creating object output



Figure 18-2

Lastly, we'll retrieve the operating system and computer name from the **Win32\_OperatingSystem** WMI class. Why do we need the computer name? Once we configure our command to run against remote computers, it's possible that the user of our tool could specify a DNS CNAME (alias) or IP address, instead of the actual computer.

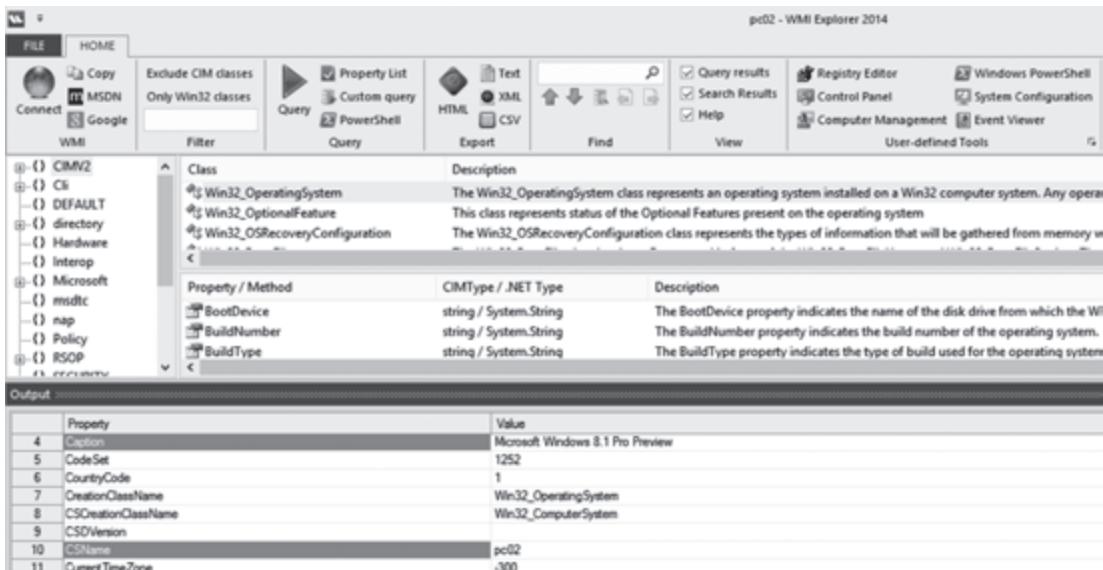


Figure 18-3

## Windows PowerShell: TFM

Now that we know where to find the required data for our system inventory tool, we'll switch over to using PowerShell Studio to create a PowerShell function that will retrieve this information.

When we create tools in PowerShell, we start out with a prototype that provides only basic functionality, so that we can proof our concept without the overhead of error checking, parameter, Help, and all of the other items that our function will eventually have. This also makes troubleshooting problems at this point much simpler.

In Figure 18-4, we've created one of these prototype functions. Although it appears that it should return the information that we're after, the results aren't as expected.

```
function Get-SystemInventory {  
    Get-WmiObject -Class Win32_OperatingSystem |  
        Select-Object CSName, Caption  
  
    Get-WmiObject -Class Win32_ComputerSystem |  
        Select-Object NumberOfLogicalProcessors, TotalPhysicalMemory  
  
    Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3' |  
        Select-Object DeviceID, Size, FreeSpace  
}
```

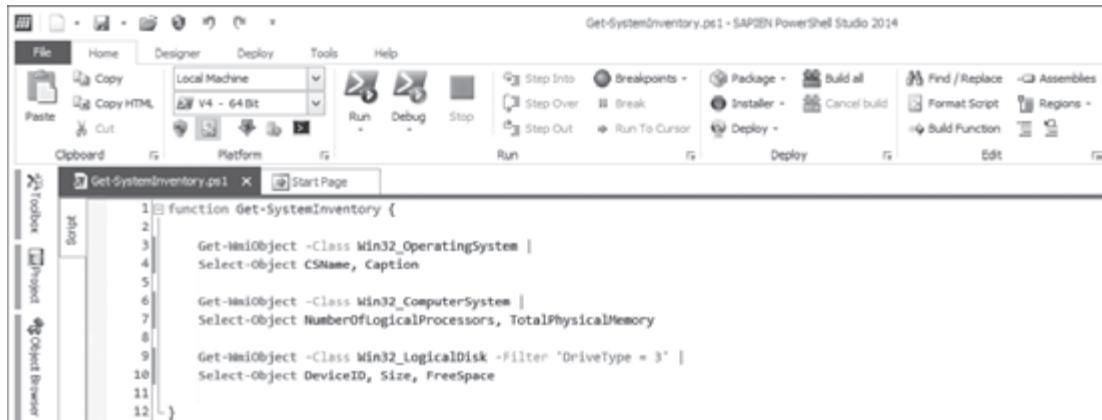


Figure 18-4

We dot-source, and then call the function shown in Figure 18-5.

```
..\Get-SystemInventory.ps1  
Get-SystemInventory
```

```
Administrator: Windows PowerShell
PS C:\Scripts> .\Get-SystemInventory.ps1
PS C:\Scripts> Get-SystemInventory
CSName
-----
pc02

Caption
-----
Microsoft Windows 8.1 Pro

PS C:\Scripts>
```

Figure 18-5

It appears to only return a portion of our data. Why does this occur? It's due to the way PowerShell's formatting system works. The formatting system looks at the first object that comes across and that object only has two properties: **CSName** and **Caption**. It has no idea that other properties will come through the pipeline later, when the other WMI queries are executed. So, it displays the output as a table, because there are less than five properties to be displayed. Then, the results of the next WMI query, which has different property names, are sent through the pipeline. However, the formatting system is only set to display **CSName** and **Caption**, which results in the properties from the other two WMI queries displaying as blank lines.

Piping the same command to **Format-List** yields different results, as shown in Figure 18-6.

```
Get-SystemInventory | Format-List
```

```
Administrator: Windows PowerShell
PS C:\Scripts> Get-SystemInventory | Format-List
CSName : pc02
Caption : Microsoft Windows 8.1 Pro
NumberOfLogicalProcessors : 4
TotalPhysicalMemory : 3220758528
DeviceID : C:
Size : 135996108800
FreeSpace : 120866336768

PS C:\Scripts>
```

Figure 18-6

## Creating a custom object

We need to combine all of the properties of the different WMI classes that we want to combine into a custom PowerShell object.

There are a number of ways to combine these items into a custom object. First, we'll start out by using a solution that's been valid since PowerShell version 1.

```
function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
```

## Windows PowerShell: TFM

```
$LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

New-Object -TypeName PSObject |
Add-Member -MemberType NoteProperty -Name 'ComputerName' -Value $OperatingSystem.CSName
-PassThru |
Add-Member -MemberType NoteProperty -Name 'OS' -Value $OperatingSystem.Caption -PassThru |
Add-Member -MemberType NoteProperty -Name 'CPULogicalCores' -Value $ComputerSystem.
NumberOfLogicalProcessors -PassThru |
Add-Member -MemberType NoteProperty -Name 'Memory' -Value $ComputerSystem.TotalPhysicalMemory
-PassThru |
Add-Member -MemberType NoteProperty -Name 'Drive' -Value $LogicalDisks.DeviceID -PassThru |
Add-Member -MemberType NoteProperty -Name 'Size' -Value $LogicalDisks.Size -PassThru |
Add-Member -MemberType NoteProperty -Name 'FreeSpace' -Value $LogicalDisks.FreeSpace -PassThru

}
```

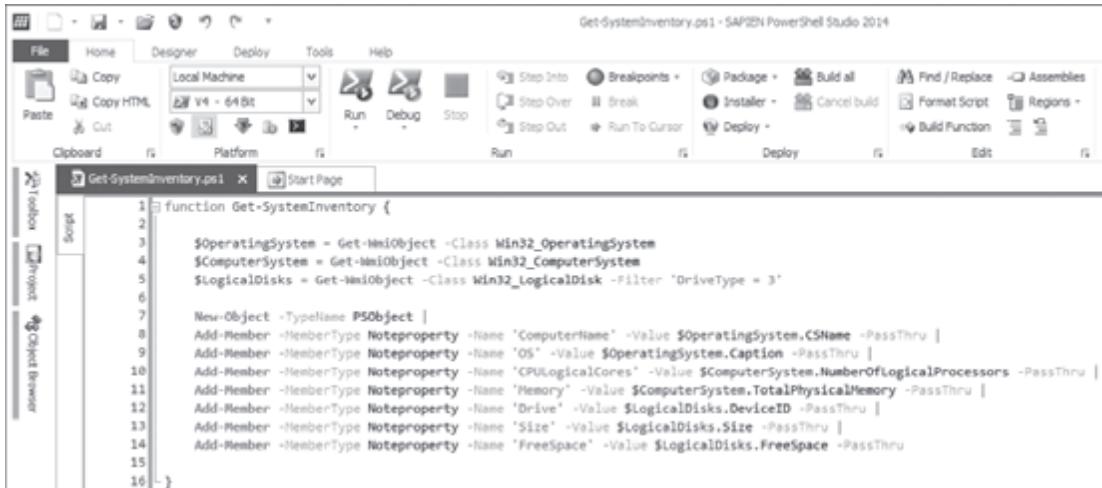
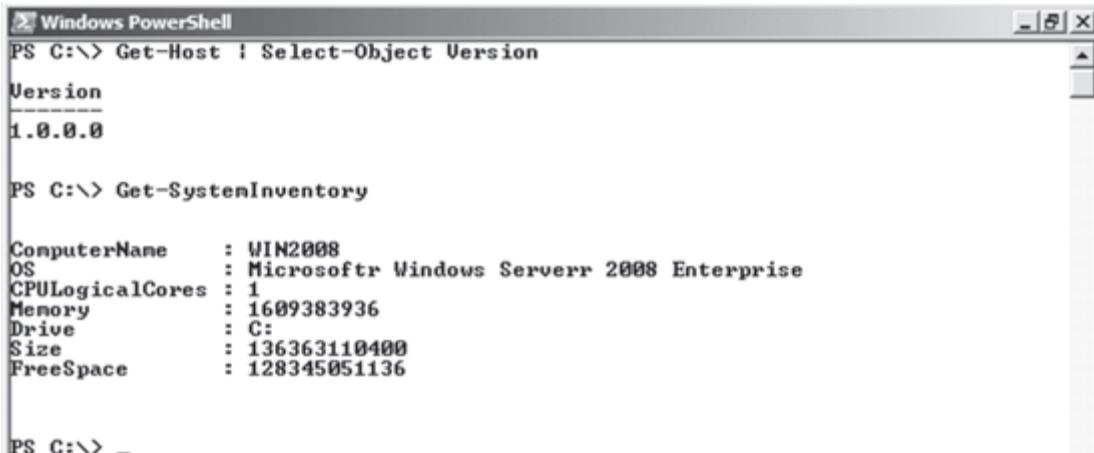


Figure 18-7

Neither of us are from the “Show-Me” state (Missouri), but we still believe in showing you, instead of just telling you, that something works on a particular version of PowerShell. As you can see in Figure 18-8, we are running this function on a Windows 2008 (non-R2) machine that only has PowerShell version 1 installed. The results of this command return the necessary requirements that we need to fulfill.

```
Get-Host | Select-Object Version
Get-SystemInventory
```



```

Windows PowerShell
PS C:\> Get-Host | Select-Object Version
Version
1.0.0.0

PS C:\> Get-SystemInventory

ComputerName      : WIN2008
OS                : Microsoft Windows Server 2008 Enterprise
CPULogicalCores   : 1
Memory            : 1689383936
Drive              : C:
Size               : 136363110400
FreeSpace          : 128345051136

PS C:\> _

```

Figure 18-8

One of the new features in PowerShell version 2 was the addition of the **-Property** parameter for the **New-Object** cmdlet, which eliminated the dependency of using **Add-Member** to add properties to a custom object.

```

function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    New-Object -TypeName PSObject -Property @{
        'ComputerName' = $OperatingSystem.CSName
        'OS' = $OperatingSystem.Caption
        'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
        'Memory' = $ComputerSystem.TotalPhysicalMemory
        'Drive' = $LogicalDisks.DeviceID
        'Size' = $LogicalDisks.Size
        'FreeSpace' = $LogicalDisks.FreeSpace
    }
}

```

## Windows PowerShell: TFM

The screenshot shows the SAPBN PowerShell Studio 2014 interface. The title bar reads "Get-SystemInventory.ps1 - SAPBN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, Help, and various developer tools like Step Into, Breakpoints, Run, Stop, and Deploy. The main area displays the PowerShell script "Get-SystemInventory.ps1". The script uses WMI objects to get system information and creates a custom object with properties like ComputerName, OS, CPULogicalCores, Memory, Drive, Size, and FreeSpace.

```
function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    New-Object -TypeName PSObject -Property @{
        'ComputerName' = $OperatingSystem.CSName
        'OS' = $OperatingSystem.Caption
        'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
        'Memory' = $ComputerSystem.TotalPhysicalMemory
        'Drive' = $LogicalDisks.DeviceID
        'Size' = $LogicalDisks.Size
        'FreeSpace' = $LogicalDisks.FreeSpace
    }
}
```

Figure 18-9

Using this method to return the results as a custom object is very similar, with the difference that they're returned in random order. This is one of the reasons we dislike this particular solution.

### Get-SystemInventory

The screenshot shows an Administrator Windows PowerShell window. The command "Get-SystemInventory" is run, and the output is displayed as a hash table. The properties and their values are: CPULogicalCores : 4, Size : 135996108800, Drive : C:, FreeSpace : 120788373504, Memory : 3220758528, ComputerName : pc02, and OS : Microsoft Windows 8.1 Pro.

```
PS C:\Scripts> Get-SystemInventory
CPULogicalCores : 4
Size            : 135996108800
Drive           : C:
FreeSpace        : 120788373504
Memory          : 3220758528
ComputerName    : pc02
OS              : Microsoft Windows 8.1 Pro

PS C:\Scripts>
```

Figure 18-10

Ordered hash tables were introduced in PowerShell version 3. It's possible to use a variation of the example shown in Figure 18-10 with an ordered hash table to solve this issue and return the results in the specified order. Be aware that there is more overhead (less efficiency) when ordering the results.

```
function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    $Properties = [ordered]@{
        'ComputerName' = $OperatingSystem.CSName
```

## Making tools: Creating object output

```
'OS' = $OperatingSystem.Caption
'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
'Memory' = $ComputerSystem.TotalPhysicalMemory
'Drive' = $LogicalDisks.DeviceID
'Size' = $LogicalDisks.Size
'FreeSpace' = $LogicalDisks.FreeSpace
}

New-Object -TypeName PSObject -Property $Properties
}
```

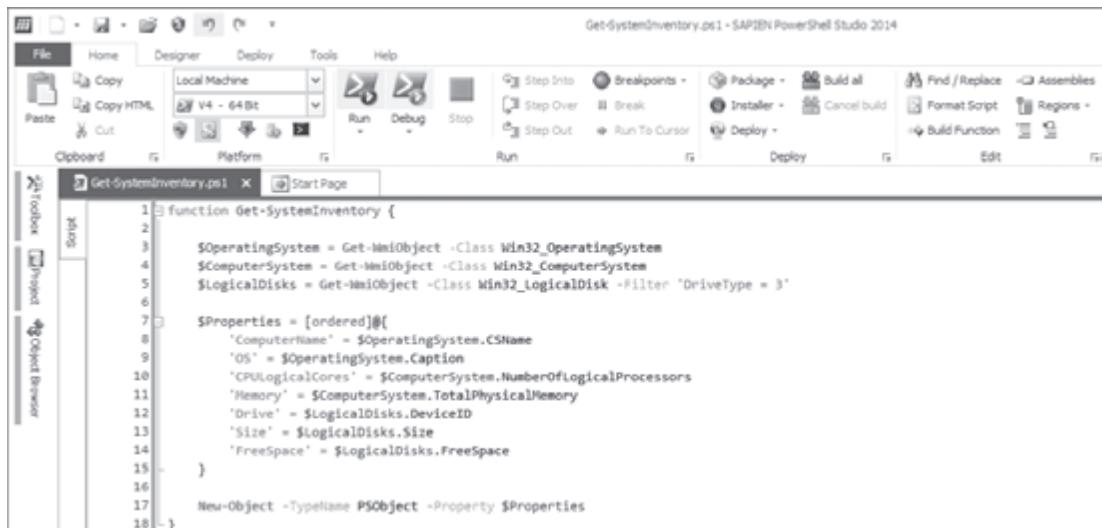
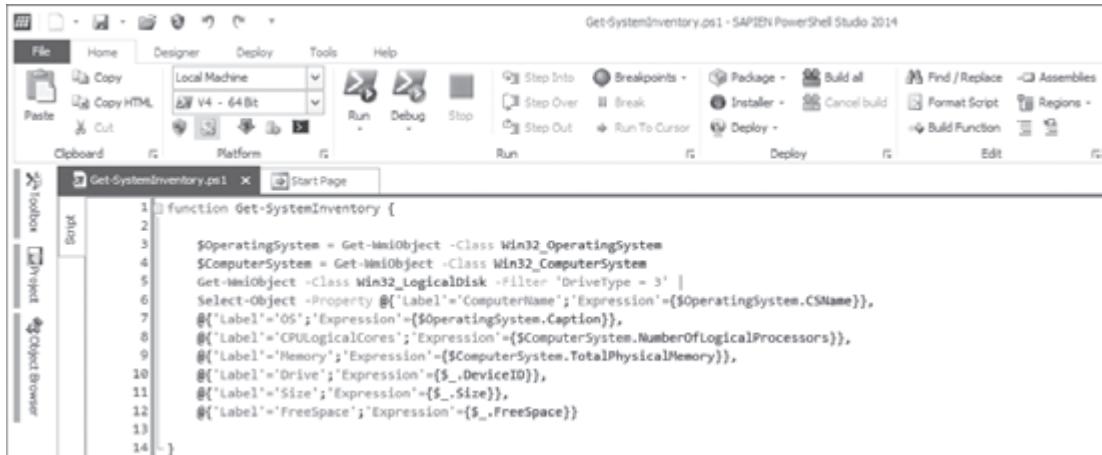


Figure 18-11

You can also use **Select-Object** to create a custom object from the properties. While this worked in previous versions of PowerShell, we like it even more in PowerShell version 3, due to the new automatic **Foreach** feature. If we have more than one logical hard drive, using this will display them on separate lines instead of displaying them as a collection. Each of the other methods would require us to add a **Foreach** loop to the code and iterate through each logical disk to display this information.

```
function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3' |
        Select-Object -Property @{
            'Label' = 'ComputerName';
            'Expression' = { $OperatingSystem.CSName
        },
        @{
            'Label' = 'OS';
            'Expression' = { $OperatingSystem.Caption } ,
        },
        @{
            'Label' = 'CPULogicalCores';
            'Expression' = { $ComputerSystem.NumberOfLogicalProcessors
        },
        @{
            'Label' = 'Memory';
            'Expression' = { $ComputerSystem.TotalPhysicalMemory } ,
        },
        @{
            'Label' = 'Drive';
            'Expression' = { $_.DeviceID } ,
            @{
                'Label' = 'Size';
                'Expression' = { $_.Size } ,
            },
            @{
                'Label' = 'FreeSpace';
                'Expression' = { $_.FreeSpace } ,
            }
        }
}
```

## Windows PowerShell: TFM



The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The title bar reads "Get-SystemInventory.ps1 - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Copy, Copy HTML, Cut, Run, Debug, Stop, Step Into, Breakpoints, Package, Build all, Find / Replace, Assemblies, Installer, Cancel build, Step Over, Step Out, Run to Cursor, Deploy, and Build Function. Below the toolbar are tabs for Clipboard, Platform, Run, Deploy, and Edit. The main area displays the PowerShell script "Get-SystemInventory.ps1". The script uses the PSCustomObject cmdlet to combine results from three WMI queries into a single object.

```
function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisk = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    [PSCustomObject]@{
        'ComputerName' = $OperatingSystem.CSName
        'OS' = $OperatingSystem.Caption
        'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
        'Memory' = $ComputerSystem.TotalPhysicalMemory
        'Drive' = $LogicalDisk.DeviceID
        'Size' = $LogicalDisk.Size
        'FreeSpace' = $LogicalDisk.FreeSpace
    }
}
```

Figure 18-12

Using the **PSCustomObject** type, which was introduced in PowerShell version 3, is our preferred method of combining the results of multiple objects into one custom object. We prefer this method because the results default to being returned in the order we specified the objects in and the performance is better.

```
function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisk = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    [PSCustomObject]@{
        'ComputerName' = $OperatingSystem.CSName
        'OS' = $OperatingSystem.Caption
        'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
        'Memory' = $ComputerSystem.TotalPhysicalMemory
        'Drive' = $LogicalDisk.DeviceID
        'Size' = $LogicalDisk.Size
        'FreeSpace' = $LogicalDisk.FreeSpace
    }
}
```

The screenshot shows the SAPiN PowerShell Studio 2014 interface. The title bar reads "Get-SystemInventory.ps1 - SAPiN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Copy, Copy HTML, Cut, Run, Debug, Stop, Step Into, Breakpoints, Package, Build all, Find / Replace, Assemblies, Format Script, Regions, and Build Function. Below the toolbar are tabs for Clipboard, Platform, Run, Deploy, and Edit. The main area displays the PowerShell script "Get-SystemInventory.ps1".

```

function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType = 3"

    [PSCustomObject]@{
        'ComputerName' = $OperatingSystem.CSName
        'OS' = $OperatingSystem.Caption
        'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
        'Memory' = $ComputerSystem.TotalPhysicalMemory
        'Drive' = $LogicalDisks.DeviceID
        'Size' = $LogicalDisks.Size
        'FreeSpace' = $LogicalDisks.FreeSpace
    }
}

```

Figure 18-13

[**PSCustomObject**] is what's called a type accelerator and the drawback to using them is that they're less discoverable when it comes to finding Help for them.

As you've seen, there are many different ways to create custom objects in PowerShell. In addition to the different ways we've shown, there are many variations of each one that can be used.

## Properties and methods

We've settled on using the example from Figure 18-13, which uses the [**PSCustomObject**] type extension to create a custom object, but we've discovered one problem with our function. It returns a collection for the hard drive values when the computer has multiple logical disks.

Get-SystemInventory

The screenshot shows an Administrator: Windows PowerShell window. The command "Get-SystemInventory" is run, and the output is displayed as a table:

	:	
ComputerName	:	pc02
OS	:	Microsoft Windows 8.1 Pro
CPULogicalCores	:	4
Memory(GB)	:	3220758528
Drive	:	{C:, D:}
Size(GB)	:	{115996108800, 136362061824}
FreeSpace(GB)	:	{120788758528, 136244940800}

Figure 18-14

In order to resolve this issue, we need to add a **Foreach** loop to iterate through each different logical disk. This should not to be confused with the **ForEach** alias of **ForEach-Object**. We typically use the **Foreach** construct in functions or scripts and the **ForEach-Object** cmdlet when we're working interactively in the PowerShell console when we're writing one-liners.

We also don't like the memory and hard drive values being returned in bytes, as shown in Figure 18-14, so we'll resolve that issue as well before continuing. As shown in Figure 18-15, we'll divide the memory by 1 GB so the results are in gigabytes and cast the result as an integer, to return a whole number with no decimal places. For the hard drive size and free space, we'll also divide them by 1 GB so our results for them are in gigabytes as well, but we'll format each of those so the results return two decimal places.

```
function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    foreach ($Disk in $LogicalDisks) {
        [PSCustomObject]@{
            ComputerName = $OperatingSystem.CSName
            'OS' = $OperatingSystem.Caption
            'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
            'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/1GB -as [int]
            'Drive' = $Disk.DeviceID
            'Size(GB)' = "{0:N2}" -f ($Disk.Size / 1GB)
            'FreeSpace(GB)' = "{0:N2}" -f ($Disk.FreeSpace / 1GB)
        }
    }
}
```



Figure 18-15

Now our results are displayed the way we want them.

`Get-SystemInventory`

```
Administrator: Windows PowerShell
PS C:\Scripts> Get-SystemInventory
ComputerName : pc02
OS           : Microsoft Windows 8.1 Pro
CPULogicalCores : 4
Memory(GB)   : 3
Drive        : C:
Size(GB)     : 126.66
FreeSpace(GB) : 112.49

ComputerName : pc02
OS           : Microsoft Windows 8.1 Pro
CPULogicalCores : 4
Memory(GB)   : 3
Drive        : D:
Size(GB)     : 127.00
FreeSpace(GB) : 126.89

PS C:\Scripts>
```

Figure 18-16

The last thing we would like is to display our results as a table by default without having to pipe to **Format-Table** each time the command is run.

## Creating custom views

By default, a list view is displayed when five or more properties are returned, unless the object type of the command has a specific default view. We're going to create a custom view for our function in order to have it default to returning a table, even though it has seven properties.

The default views that PowerShell ships with are located in the \$PSHOME folder.

`Get-ChildItem -Path $PSHOME\*format.ps1xml`

```
Administrator: Windows PowerShell
PS C:\> Get-ChildItem -Path $PSHOME\*format.ps1xml
Directory: C:\Windows\System32\WindowsPowerShell\v1.0

Mode                LastWriteTime      Length Name
----              -----          ---- -----
-a---  2/28/2013 4:10 PM       27338 Certificate.format.ps1xml
-a---  2/28/2013 4:10 PM       27106 Diagnostics.Format.ps1xml
-a---  4/4/2013  6:56 PM      147702 DotNetTypes.format.ps1xml
-a---  2/28/2013 4:10 PM       14502 Event.Format.ps1xml
-a---  2/28/2013 4:10 PM       21293 FileSystem.format.ps1xml
-a---  2/28/2013 4:10 PM      287938 Help.format.ps1xml
-a---  2/28/2013 4:10 PM       97880 HelpV3.format.ps1xml
-a---  4/4/2013  6:56 PM      103082 PowerShellCore.format.ps1xml
-a---  2/28/2013 4:10 PM      18612 PowerShellTrace.format.ps1xml
-a---  2/28/2013 4:10 PM       13659 Registry.format.ps1xml
-a---  2/28/2013 4:10 PM      17731 WSHan.Format.ps1xml

PS C:\>
```

Figure 18-17

## Windows PowerShell: TFM

These files are digitally signed, so we're not going to open them because any modification, even accidental, will render these files useless. We'll copy the DotNetTypes.Format.ps1xml file to our scripts folder and name it SystemInventory.format.ps1xml.

```
Copy-Item -Path $PSHOME\DotNetTypes.format.ps1xml -Destination C:\Scripts\SystemInventory.format.ps1xml
```



Figure 18-18

We'll open the copy of the file (C:\Scripts\SystemInventory.format.ps1xml) in PrimalXML 2014. Remove everything except the first line, leaving the opening and closing **Configuration** and **ViewDefinitions** tags. We'll also leave the entire **System.Globalization.CultureInfo** structure in place.

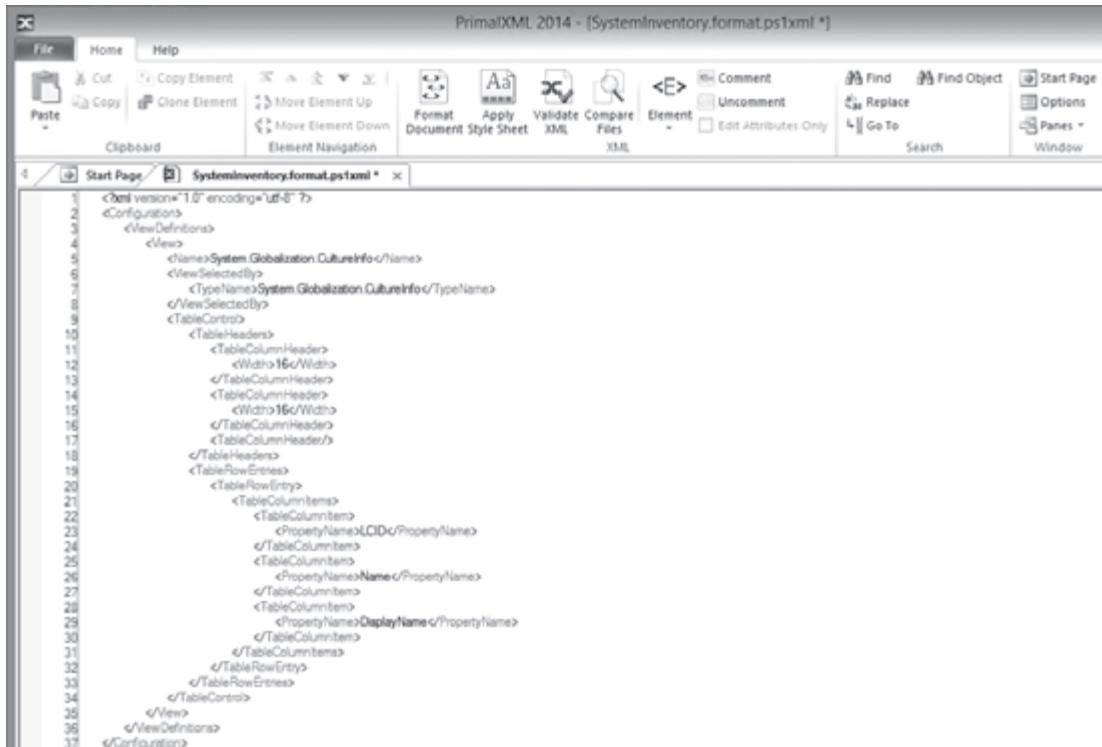


Figure 18-19

We'll change the value in the **<Name>** tag to **TFM.SystemInventory**, so you'll know that these are

from this TFM book. We'll use that same name for the value of the `<TypeName>` tag, which tells PowerShell what type of object that these formatting rules apply to. We'll specify a total of seven column headers, each with a custom width value and one for each property returned by our `Get-SystemInventory` function. We'll also specify seven property names in the `TableRowEntries` section.

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <ViewDefinitions>
    <View>
      <Name>TFM.SystemInventory</Name>
      <ViewSelectedBy>
        <TypeName>TFM.SystemInventory</TypeName>
      </ViewSelectedBy>
      <TableControl>
        <TableHeaders>
          <TableColumnHeader>
            <Width>14</Width>
          </TableColumnHeader>
          <TableColumnHeader>
            <Width>40</Width>
          </TableColumnHeader>
          <TableColumnHeader>
            <Width>17</Width>
          </TableColumnHeader>
          <TableColumnHeader>
            <Width>12</Width>
          </TableColumnHeader>
          <TableColumnHeader>
            <Width>7</Width>
          </TableColumnHeader>
          <TableColumnHeader>
            <Width>10</Width>
          </TableColumnHeader>
          <TableColumnHeader>
            <Width>15</Width>
          </TableColumnHeader>
        </TableHeaders>
        <TableRowEntries>
          <TableRowEntry>
            <TableColumnItems>
              <TableColumnItem>
                <PropertyName>ComputerName</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>OS</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>CPULogicalCores</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>Memory(GB)</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>Drive</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>Size(GB)</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>FreeSpace(GB)</PropertyName>
              </TableColumnItem>
            </TableColumnItems>
          </TableRowEntry>
        </TableRowEntries>
      </TableControl>
    </View>
  </ViewDefinitions>
</Configuration>
```

## Windows PowerShell: TFM

```
</TableRowEntry>
</TableRowEntries>
</TableControl>
</View>
</ViewDefinitions>
</Configuration>
```

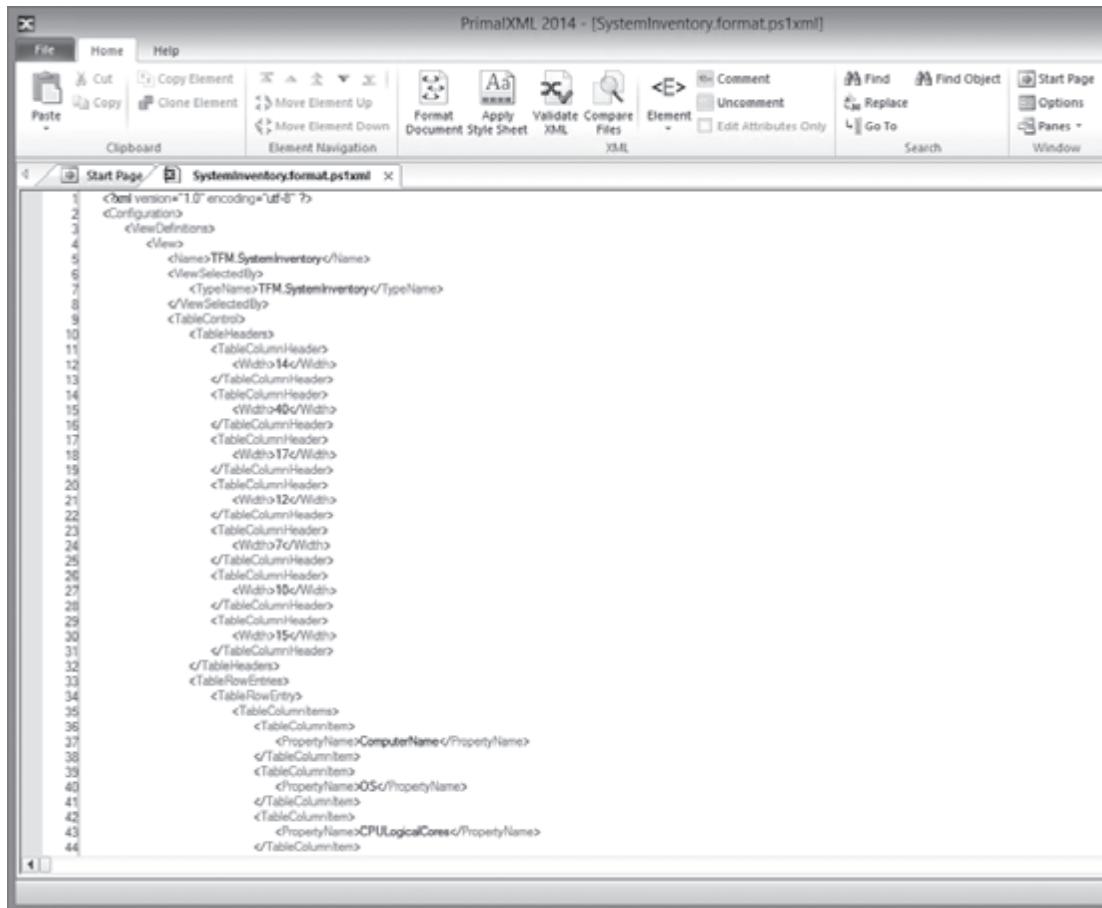


Figure 18-20

Currently our function produces a **PSCustomObject** type of object.

```
Get-SystemInventory | Get-Member
```

```

TypeName: System.Management.Automation.PSCustomObject
Name      MemberType  Definition
----      ----       -----
Equals    Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType   Method     type GetType()
ToString  Method     string ToString()
ComputerName NoteProperty System.String ComputerName=pc02
CPULogicalCores NoteProperty System.UInt32 CPULogicalCores=4
Drive     NoteProperty System.String Drive=C:
FreeSpace(GB) NoteProperty System.String FreeSpace(GB)=112.08
Memory(GB)  NoteProperty System.Int32 Memory(GB)=3
OS        NoteProperty System.String OS=Microsoft Windows 8.1 Pro
Size(GB)   NoteProperty System.String Size(GB)=126.66

```

Figure 18-21

Based on the value of <TypeName> that we used in our SystemInventory.format.ps1xml file, we need it to produce a **TFM.SystemInventory** type object, so we'll go back to our script in PowerShell Studio and make a couple of modifications to accommodate this.

```

function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

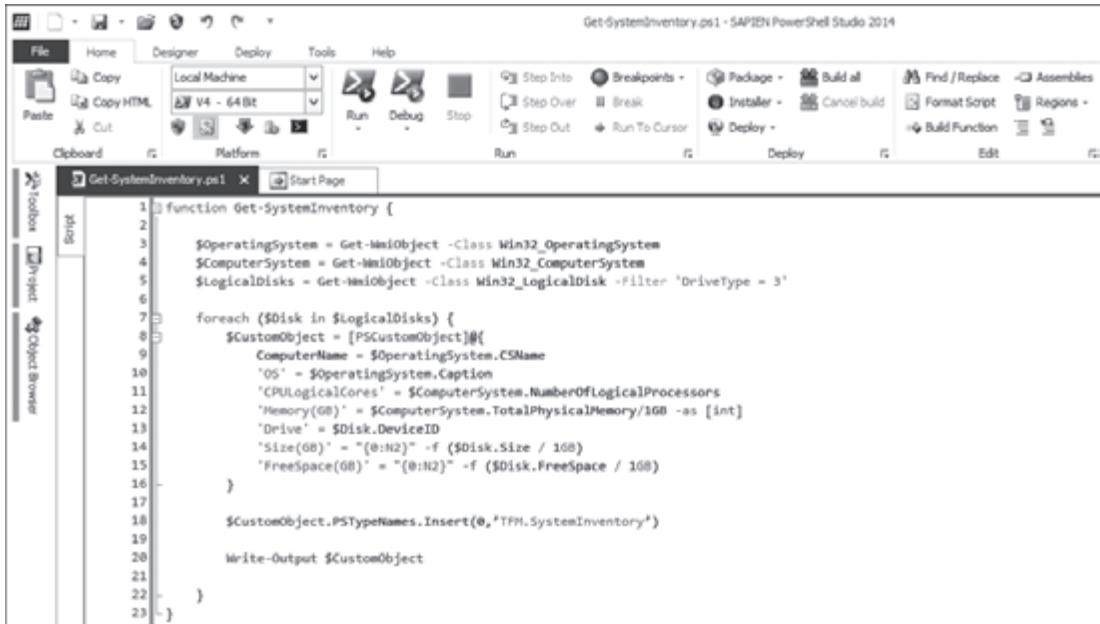
    foreach ($Disk in $LogicalDisks) {
        $CustomObject = [PSCustomObject]@{
            ComputerName = $OperatingSystem.CSName
            'OS' = $OperatingSystem.Caption
            'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
            'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/1GB -as [int]
            'Drive' = $Disk.DeviceID
            'Size(GB)' = "{0:N2}" -f ($Disk.Size / 1GB)
            'FreeSpace(GB)' = "{0:N2}" -f ($Disk.FreeSpace / 1GB)
        }

        $CustomObject.PSTypeNames.Insert(0,'TFM.SystemInventory')

        Write-Output $CustomObject
    }
}

```

## Windows PowerShell: TFM



The screenshot shows the SAPiEN PowerShell Studio interface. The title bar reads "Get-SystemInventory.ps1 - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Copy, Copy HTML, Cut, Run, Debug, Stop, Step Into, Breakpoints, Package, Build all, Find / Replace, Assemblies, Installer, Cancel build, Deploy, and Build Function. Below the toolbar are tabs for Clipboard, Platform, Run, Deploy, and Edit. The main area displays the PowerShell script "Get-SystemInventory.ps1". The script defines a function "Get-SystemInventory" that retrieves system information using WMI objects and creates a custom object with properties like ComputerName, OS, CPULogicalCores, Memory(GB), Drive, FreeSpace(GB), and OS.

```
function Get-SystemInventory {
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType = 3"

    foreach ($Disk in $LogicalDisks) {
        $CustomObject = [PSCustomObject]@{
            ComputerName = $OperatingSystem.CSName
            'OS' = $OperatingSystem.Caption
            'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
            'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/1GB -as [int]
            'Drive' = $Disk.DeviceID
            'Size(GB)' = "{0:N2}" -f ($Disk.Size / 1GB)
            'FreeSpace(GB)' = "{0:N2}" -f ($Disk.FreeSpace / 1GB)
        }

        $CustomObject.PSTypeNames.Insert(0,"TFM.SystemInventory")
    }

    Write-Output $CustomObject
}
```

Figure 18-22

Now we're producing a **TFM.SystemInventory** type object.

```
Get-SystemInventory | Get-Member
```



The screenshot shows an Administrator Windows PowerShell window. The command "Get-SystemInventory | Get-Member" is run. The output shows the type definition for "TFM.SystemInventory" with properties: Equals, GetHashCode, GetType, ToString, ComputerName, CPULogicalCores, Drive, FreeSpace(GB), Memory(GB), OS, and Size(GB). Each property is defined as a NoteProperty of type System.String or System.UInt32.

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
ComputerName	NoteProperty	System.String ComputerName=pc02
CPULogicalCores	NoteProperty	System.UInt32 CPULogicalCores=4
Drive	NoteProperty	System.String Drive=C:
FreeSpace(GB)	NoteProperty	System.String FreeSpace(GB)=112.08
Memory(GB)	NoteProperty	System.Int32 Memory(GB)=3
OS	NoteProperty	System.String OS=Microsoft Windows 8.1 Pro
Size(GB)	NoteProperty	System.String Size(GB)=126.66

Figure 18-23

We load the view definition, but loading it this way will only exist for the life of our PowerShell session, so this is primarily a way to test the functionality.

```
Update-FormatData -PrependPath .\SystemInventory.format.ps1xml
```

## Making tools: Creating object output

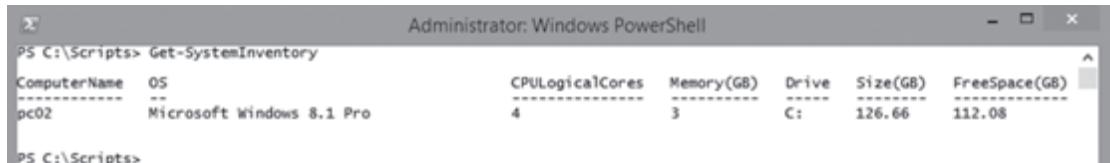


```
Administrator: Windows PowerShell
PS C:\Scripts> Update-FormatData -PrependPath .\SystemInventory.format.ps1xml
PS C:\Scripts>
```

Figure 18-24

Now our **Get-SystemInventory** function defaults to returning its output as a table.

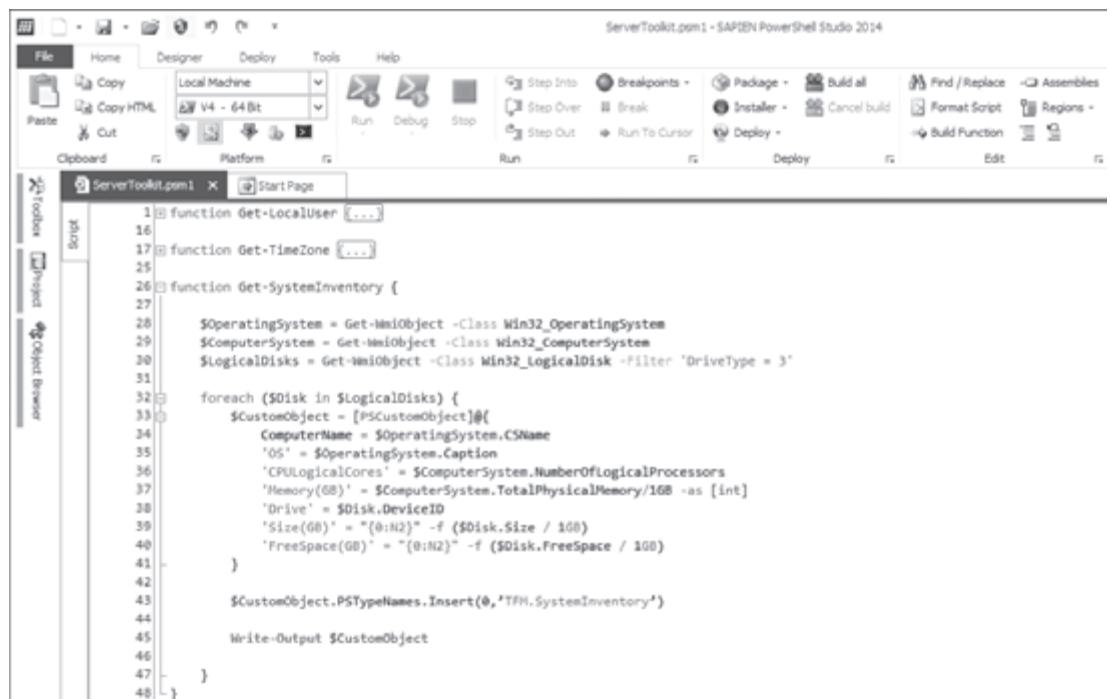
**Get-SystemInventory**



```
Administrator: Windows PowerShell
PS C:\Scripts> Get-SystemInventory
ComputerName OS CPULogicalCores Memory(GB) Drive Size(GB) FreeSpace(GB)
pc02 Microsoft Windows 8.1 Pro 4 3 C: 126.66 112.08
PS C:\Scripts>
```

Figure 18-25

We're now adding our **Get-SystemInventory** function to our **ServerToolkit** module that we created in a previous chapter.



```
ServerToolkit.psm1 - SAPiEN PowerShell Studio 2014

File Home Deploy Tools Help
Copy Local Machine V4 - 64 Bit Run Debug Stop
Copy HTML Cut Step Into Breakpoints Package Installer Cancel build
Clipboard Platform Step Over Break Step Out Run To Cursor Deploy
Run Deploy Edit
Start Page

ServerToolkit.psm1
1 function Get-LocalUser { ... }
16
17 function Get-TimeZone { ... }
25
26 function Get-SystemInventory {
27
28     $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
29     $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
30     $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'
31
32     foreach ($Disk in $LogicalDisks) {
33         $CustomObject = [PSCustomObject]@{
34             ComputerName = $OperatingSystem.CSName
35             'OS' = $OperatingSystem.Caption
36             'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
37             'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/1GB -as [int]
38             'Drive' = $Disk.DeviceID
39             'Size(GB)' = "{0:N2}" -f ($Disk.Size / 1GB)
40             'FreeSpace(GB)' = "{0:N2}" -f ($Disk.FreeSpace / 1GB)
41         }
42
43         $CustomObject.PSTypeNames.Insert(0,'TFM.SystemInventory')
44
45         Write-Output $CustomObject
46
47     }
48 }
```

Figure 18-26

We can see that this function is now part of that module.

```
Get-Command -Module ServerToolkit
```

CommandType	Name	ModuleName
Function	Get-LocalUser	ServerToolkit
Function	Get-SystemInventory	ServerToolkit
Function	Get-TimeZone	ServerToolkit

Figure 18-27

We've copied our SystemInventory.format.ps1xml file into our module folder for the **ServerToolkit** module and renamed it to ServerToolkit.format.ps1xml, since this is the custom view file that we'll use for other functions in this module as well.

We'll modify the Module manifest file so it knows about the custom view.

```
#
# Module manifest for module 'ServerToolkit'
#
@{
    # Script module or binary module file associated with this manifest.
    RootModule = 'ServerToolkit'

    # Version number of this module.
    ModuleVersion = '1.0'

    # ID used to uniquely identify this module
    GUID = 'cf87135c-c023-47dc-9b01-a2a5b2ee0f3d'

    # Author of this module
    Author = 'administrator'

    # Company or vendor of this module
    CompanyName = 'Unknown'

    # Copyright statement for this module
    Copyright = '(c) 2014 administrator. All rights reserved.'

    # Description of the functionality provided by this module
    # Description = ''

    # Minimum version of the Windows PowerShell engine required by this module
    PowerShellVersion = '4.0'

    # Name of the Windows PowerShell host required by this module
    # PowerShellHostName = ''

    # Minimum version of the Windows PowerShell host required by this module
    # PowerShellHostVersion = ''
```

```
# Minimum version of Microsoft .NET Framework required by this module
# DotNetFrameworkVersion = ''

# Minimum version of the common language runtime (CLR) required by this module
# CLRVersion = ''

# Processor architecture (None, X86, Amd64) required by this module
# ProcessorArchitecture = ''

# Modules that must be imported into the global environment prior to importing this module
# RequiredModules = @()

# Assemblies that must be loaded prior to importing this module
# RequiredAssemblies = @()

# Script files (.ps1) that are run in the caller's environment prior to importing this module.
# ScriptsToProcess = @()

# Type files (.ps1xml) to be loaded when importing this module
# TypesToProcess = @()

# Format files (.ps1xml) to be loaded when importing this module
FormatsToProcess = 'ServerToolkit.format.ps1xml'

# Modules to import as nested modules of the module specified in RootModule/ModuleToProcess
# NestedModules = @()

# Functions to export from this module
FunctionsToExport = '*'

# Cmdlets to export from this module
CmdletsToExport = '*'

# Variables to export from this module
VariablesToExport = '*'

# Aliases to export from this module
AliasesToExport = '*'

# List of all modules packaged with this module
# ModuleList = @()

# List of all files packaged with this module
# FileList = @()

# Private data to pass to the module specified in RootModule/ModuleToProcess
# PrivateData = ''

# HelpInfo URI of this module
# HelpInfoURI = ''

# Default prefix for commands exported from this module. Override the default prefix using Import-Module -Prefix.
# DefaultCommandPrefix = ''

}
```

## Windows PowerShell: TFM

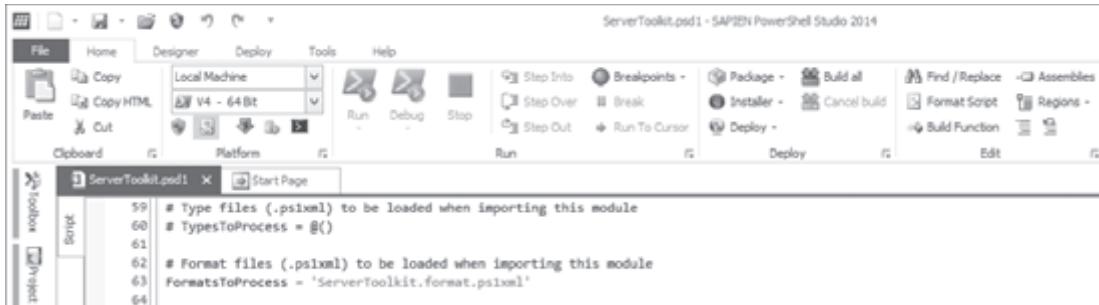


Figure 18-28

We can provide this module to our tool users and their results from the **Get-SystemInventory** function will automatically be returned in table format.

We're now going to run our **Get-SystemInventory** function, which only exists on our local computer, against the servers in our environment to complete the task that we originally set out to accomplish.

```
Invoke-Command -ComputerName dc01, sql01, web01 -ScriptBlock ${function:Get-SystemInventory}
```

A screenshot of an Administrator Windows PowerShell window titled "Administrator: Windows PowerShell". The command run is "PS C:\Scripts> Invoke-Command -ComputerName dc01, sql01, web01 -ScriptBlock \${function:Get-SystemInventory}". The output is a table:

ComputerName	OS	CPU	LogicalCores	Memory(GB)	Drive	Size(GB)	FreeSpace(GB)
DC01	Microsoft Windows Server 2012 R2 Data...	2	1	126.66	C:	121.27	
SQL01	Microsoft Windows Server 2012 R2 Data...	2	1	126.66	C:	120.12	
WEB01	Microsoft Windows Server 2012 R2 Data...	2	1	126.66	C:	122.27	

Figure 18-29

This is really only the tip of the iceberg when it comes to custom views. If this is something you're interested in learning more about, we recommend viewing the `about_Format.ps1xml` and `about_Types.ps1xml` Help files from within PowerShell. MSDN is another good resource for learning more about custom views in PowerShell. We'll refer you to two specific MSDN articles that contain information about what we've covered in this chapter: "How to Create a Formatting File (.format.ps1xml)": <http://tinyurl.com/k6yllbc> and "Creating a Table View": <http://tinyurl.com/k7h9eoy>.

## Exercise 18 – Creating custom output

The junior-level admins like the reusable tool that you created in the exercise from chapter 16 so much that they've asked you to make some modifications to it so that it retrieves additional information from the remote computers that they support.

Begin with the function that you created in the exercise from chapter 16.

### Task 1

Add the manufacturer and model of the computer to the results of your function. This information can be found in the **Win32\_ComputerSystem** WMI class.

### Task 2

Create a custom object that combines the output of both WMI classes into a single object.



## Chapter 19

# Making tools: Adding Help

### Why should I add Help?

What method do you use to determine how to use commands in PowerShell? Most people simply run **Get-Help** followed by the command name, and for more detailed information they specify one of the **Get-Help** parameters, such as **-Full**, **-Detailed**, **-Online**, or **-ShowWindow**. Wouldn't it be nice if the functions and scripts created by third-party toolmakers, such as you, could use this same sort of standardized Help so that users could discover how to use these third-party commands just as they do the native commands that are built into PowerShell?

You're in luck! PowerShell offers just that—the ability to add Help to your functions and scripts. We believe that if your PowerShell code is worth saving as a script or function, then it should have Help included with it.

### The Help documentation for the operator

#### Comment-based Help

Comments in PowerShell scripts and functions begin with an octothorpe, or pound sign (#), and while it's possible to add a pound sign to the beginning of every line in comment-based Help, it's our recommendation to use block comments, which begin with a less than sign, followed by a pound sign (<#>), and then ends with a pound sign, followed by a greater than sign (#>).

In our opinion, comment-based Help is the de facto standard for PowerShell toolmakers. By adding comment-based Help, the users of your scripts and functions will be able to use the **Get-Help** cmdlet to determine how to use them, just as they would any other PowerShell command.

The easiest way to add comment-based Help is to use Ctrl+K to bring up code snippets from within PowerShell Studio.

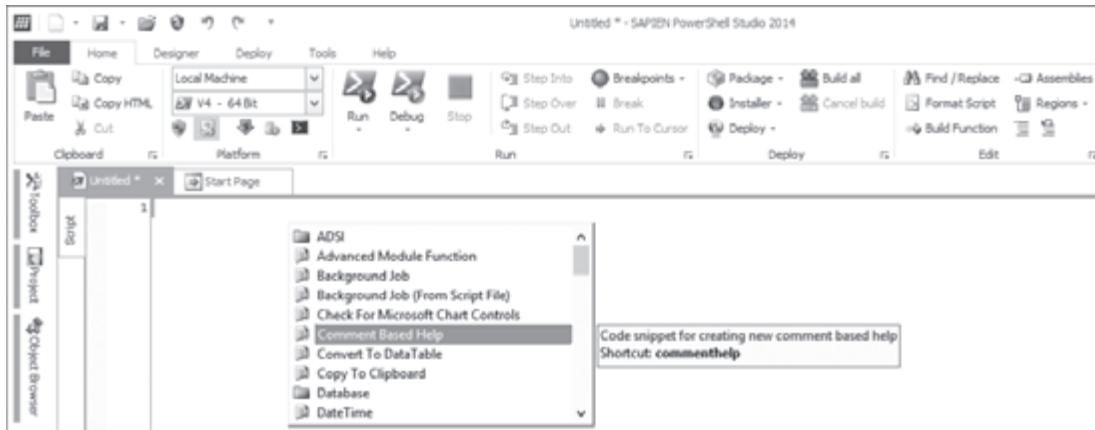
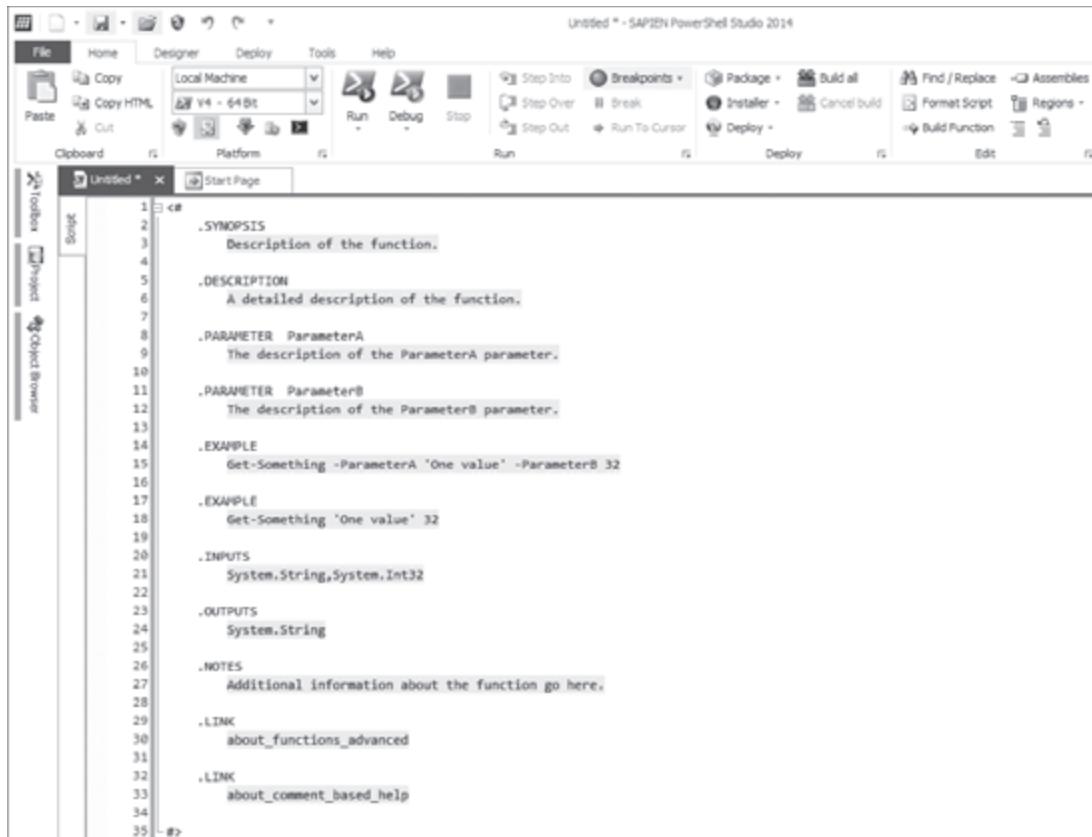


Figure 19-1

Select the **Comment Based Help** snippet, as shown in Figure 19-1, and then press **Enter**. The template for comment-based Help will be added beginning at the current position where the cursor resides.



The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The title bar reads "Untitled \* - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Copy, Copy HTML, Cut, Paste, Run, Debug, Stop, Step Into, Breakpoints, Package, Build all, Find / Replace, Assemblies, Installer, Cancel build, Deploy, and Format Script. The main window displays a PowerShell script titled "Untitled \*". The script content is as follows:

```

1 <#
2     .SYNOPSIS
3         Description of the function.
4
5     .DESCRIPTION
6         A detailed description of the function.
7
8     .PARAMETER ParameterA
9         The description of the ParameterA parameter.
10
11    .PARAMETER ParameterB
12        The description of the ParameterB parameter.
13
14    .EXAMPLE
15        Get-Something -ParameterA 'One value' -ParameterB 32
16
17    .EXAMPLE
18        Get-Something 'One value' 32
19
20    .INPUTS
21        System.String, System.Int32
22
23    .OUTPUTS
24        System.String
25
26    .NOTES
27        Additional information about the function go here.
28
29    .LINK
30        about_functions_advanced
31
32    .LINK
33        about_comment_based_help
34
35 -#>

```

Figure 19-2

As shown in Figure 19-2, comment-based Help uses a series of keywords that each begin with a dot (.). Although the code snippet added them in upper case, and they're normally seen used in upper case, they're not case sensitive. To use comment-based Help, at least one keyword must be defined and some of the keywords, such as "parameter", can be used multiple times.

## Comment-based Help for functions

When used with a function, comment-based Help must be placed in one of three locations:

At the beginning of the function body. We've listed this option first because this option is our recommendation for the most standardized location to place your comment-based Help within a function. Figure 19-3 shows a function named **Test-Port** that we've added comment-based Help to by using this method.

```
#Requires -Version 4.0
function Test-Port {

<#
.SYNOPSIS
Determines if a port is open on a computer.
.DESCRIPTION
Test-Port is a function used to determine if a port is open on a local or remote computer.
```

## Windows PowerShell: TFM

```

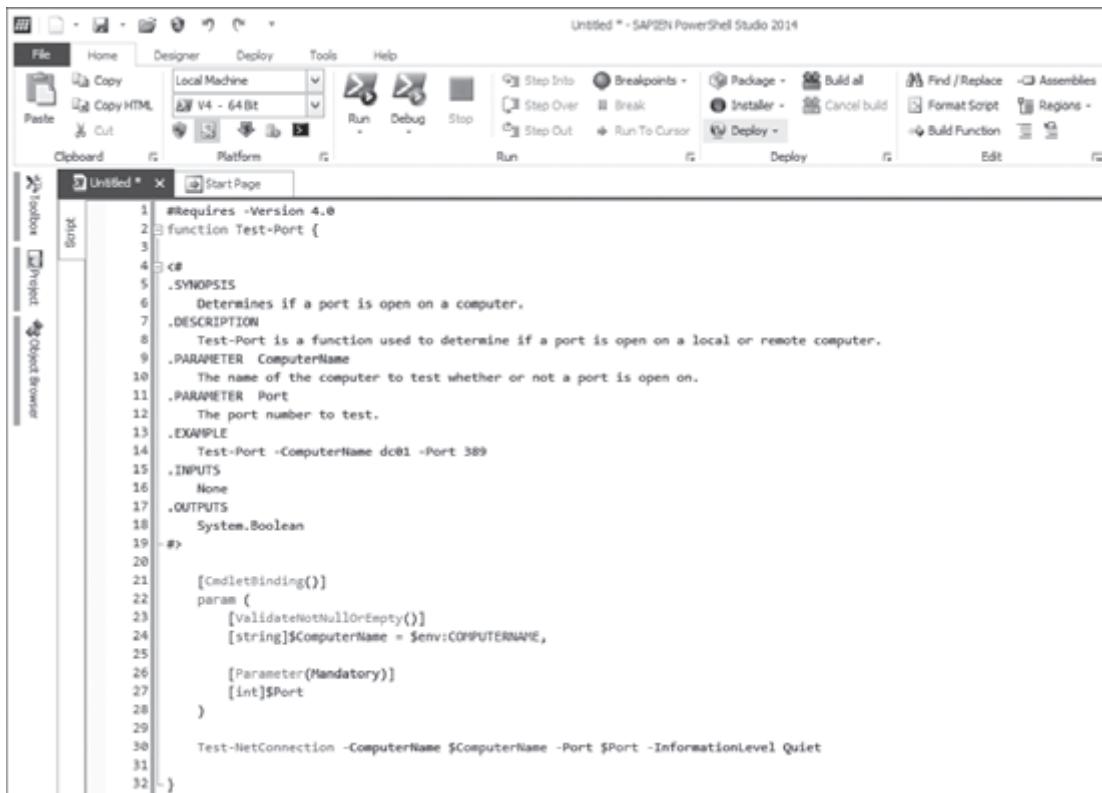
.PARAMETER ComputerName
    The name of the computer to test whether or not a port is open on.
.PARAMETER Port
    The port number to test.
.EXAMPLE
    Test-Port -ComputerName dc01 -Port 389
.INPUTS
    None
.OUTPUTS
    System.Boolean
#>

[CmdletBinding()]
param (
    [ValidateNotNullOrEmpty()]
    [string]$ComputerName = $env:COMPUTERNAME,

    [Parameter(Mandatory)]
    [int]$Port
)

Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
}

```



The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The main window displays a PowerShell script for a function named `Test-Port`. The script includes documentation blocks (.PARAMETER, .DESCRIPTION, .EXAMPLE), input parameters, and a cmdlet binding block. The script uses the `Test-NetConnection` cmdlet to check if a port is open. The interface includes a toolbar with various icons for file operations, deployment, and debugging.

```

#Requires -Version 4.0
function Test-Port {
    ##
    .SYNOPSIS
        Determines if a port is open on a computer.
    .DESCRIPTION
        Test-Port is a function used to determine if a port is open on a local or remote computer.
    .PARAMETER ComputerName
        The name of the computer to test whether or not a port is open on.
    .PARAMETER Port
        The port number to test.
    .EXAMPLE
        Test-Port -ComputerName dc01 -Port 389
    .INPUTS
        None
    .OUTPUTS
        System.Boolean
#>

[CmdletBinding()]
param (
    [ValidateNotNullOrEmpty()]
    [string]$ComputerName = $env:COMPUTERNAME,

    [Parameter(Mandatory)]
    [int]$Port
)

Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
}

```

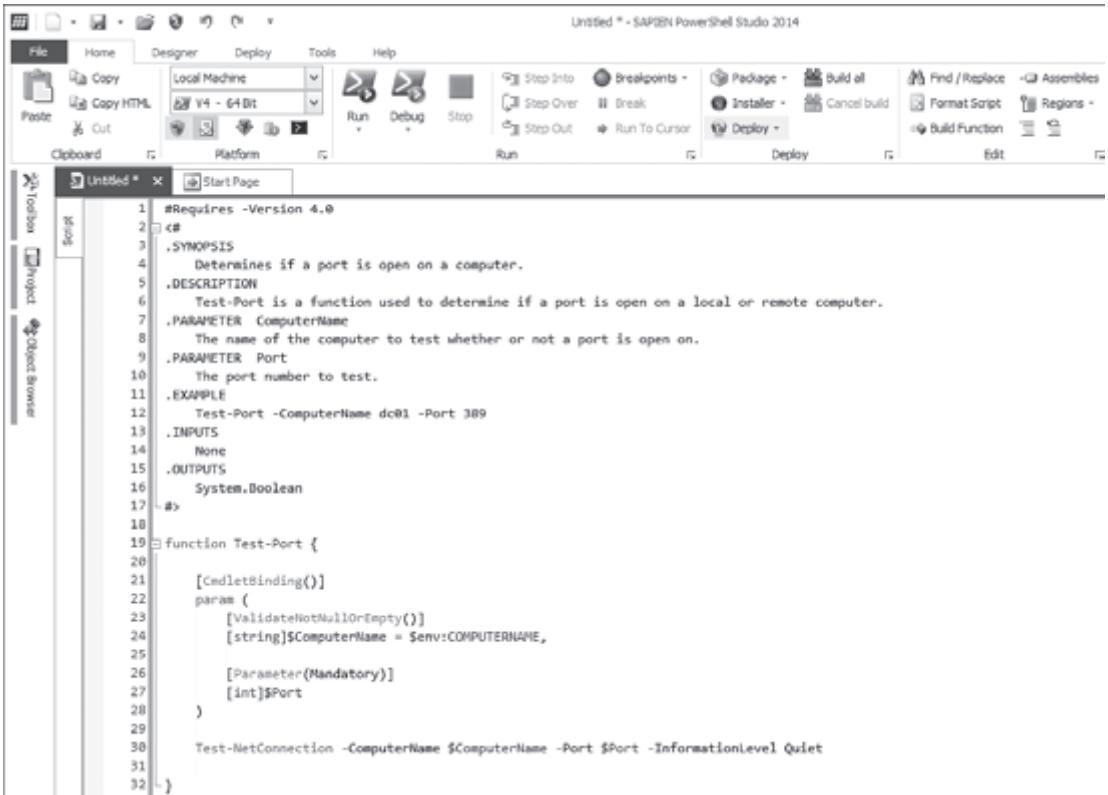
Figure 19-3

One of the other valid locations that comment-based Help for functions can be placed is before the **Function** keyword. If you choose to place your comment-based Help before the **Function** keyword, there should be no more than one empty line between the last line of the Help and the line where the function keyword is specified.

```
#Requires -Version 4.0
<#
.SYNOPSIS
    Determines if a port is open on a computer.
.DESCRIPTION
    Test-Port is a function used to determine if a port is open on a local or remote computer.
.PARAMETER ComputerName
    The name of the computer to test whether or not a port is open on.
.PARAMETER Port
    The port number to test.
.EXAMPLE
    Test-Port -ComputerName dc01 -Port 389
.INPUTS
    None
.OUTPUTS
    System.Boolean
#>

function Test-Port {
    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string]$ComputerName = $env:COMPUTERNAME,
        [Parameter(Mandatory)]
        [int]$Port
    )
    Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
}
```

## Windows PowerShell: TFM



```

#Requires -Version 4.0
<#
.c#
.SYNOPSIS
    Determines if a port is open on a computer.
.DESCRIPTION
    Test-Port is a function used to determine if a port is open on a local or remote computer.
.PARAMETER ComputerName
    The name of the computer to test whether or not a port is open on.
.PARAMETER Port
    The port number to test.
.EXAMPLE
    Test-Port -ComputerName dc01 -Port 389
.INPUTS
    None
.OUTPUTS
    System.Boolean
#>
function Test-Port {
    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string]$ComputerName = $env:COMPUTERNAME,
        [Parameter(Mandatory)]
        [int]$Port
    )
    Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
}

```

Figure 19-4

The other valid location for comment-based Help in a function is at the end of the function body.

```

#Requires -Version 4.0

function Test-Port {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string]$ComputerName = $env:COMPUTERNAME,

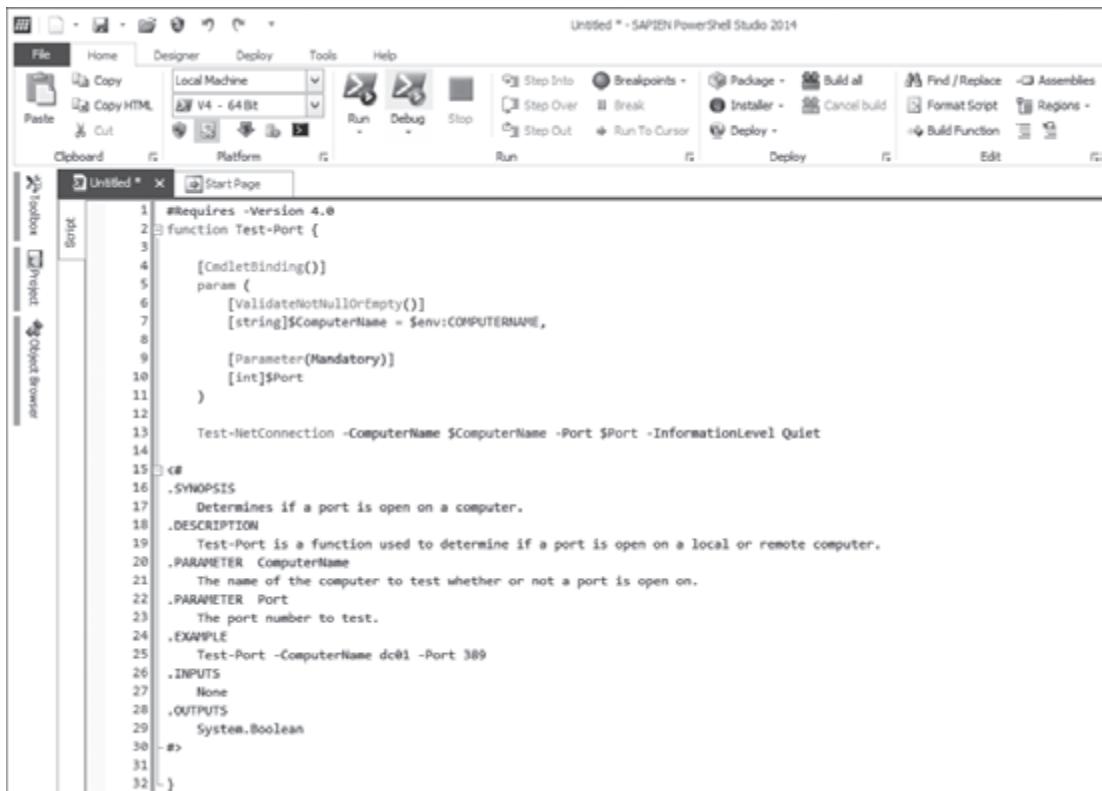
        [Parameter(Mandatory)]
        [int]$Port
    )
    Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet

<#
.SYNOPSIS
    Determines if a port is open on a computer.
.DESCRIPTION
    Test-Port is a function used to determine if a port is open on a local or remote computer.
.PARAMETER ComputerName
    The name of the computer to test whether or not a port is open on.
.PARAMETER Port
    The port number to test.

```

## Making tools: Adding Help

```
.EXAMPLE
    Test-Port -ComputerName dc01 -Port 389
.INPUTS
    None
.OUTPUTS
    System.Boolean
#>
}
```



The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The main window displays a PowerShell script named 'Untitled.ps1'. The script contains a function 'Test-Port' with parameters for ComputerName and Port, and a description of its purpose, parameters, and examples. The interface includes a toolbar with various icons for file operations, deployment, and debugging. The status bar at the bottom indicates 'Untitled \* - SAPiEN PowerShell Studio 2014'.

```
1 #Requires -Version 4.0
2 function Test-Port {
3
4     [CmdletBinding()]
5     param (
6         [ValidateNotNullOrEmpty()]
7         [string]$ComputerName = $env:COMPUTERNAME,
8
9         [Parameter(Mandatory)]
10        [int]$Port
11    )
12
13    Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
14
15    #
16    .SYNOPSIS
17        Determines if a port is open on a computer.
18    .DESCRIPTION
19        Test-Port is a function used to determine if a port is open on a local or remote computer.
20    .PARAMETER ComputerName
21        The name of the computer to test whether or not a port is open on.
22    .PARAMETER Port
23        The port number to test.
24    .EXAMPLE
25        Test-Port -ComputerName dc01 -Port 389
26    .INPUTS
27        None
28    .OUTPUTS
29        System.Boolean
30    #>
31
32 }
```

Figure 19-5

Regardless of which location is chosen, after dot-sourcing the script containing our function, we can use **Get-Help**, with the **Test-Port** function, just like we would any other PowerShell command.

```
..\Test-Port.ps1
Get-Help -Name Test-Port -Full
```



```

Administrator: Windows PowerShell
PS C:\Scripts> . .\Test-Port.ps1
PS C:\Scripts> Get-Help -Name Test-Port -Full

NAME
    Test-Port

SYNOPSIS
    Determines if a port is open on a computer.

SYNTAX
    Test-Port [[-ComputerName] <String>] [-Port] <Int32> [<CommonParameters>]

DESCRIPTION
    Test-Port is a function used to determine if a port is open on a local or remote computer.

PARAMETERS
    -ComputerName <String>
        The name of the computer to test whether or not a port is open on.

        Required?           false
        Position?          1
        Default value      $env:COMPUTERNAME
        Accept pipeline input?  false
        Accept wildcard characters?  false

    -Port <Int32>
        The port number to test.

        Required?           true
        Position?          2
        Default value      0
        Accept pipeline input?  false
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    None

OUTPUTS
    System.Boolean

-----
EXAMPLE 1 -----
C:\PS>Test-Port -ComputerName dc01 -Port 389

```

Figure 19-6

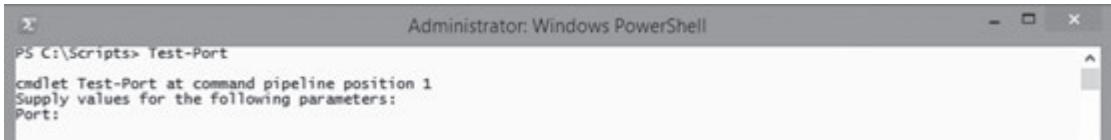
## Comment-based Help for scripts

Comment-based Help for scripts works similar to the way it does for functions, except it can only be added in one of two locations:

- **At the beginning of the script.** This is what we recommend for scripts, because it's the most trouble-free location for adding command-based Help to scripts.
- **At the end of the script.** If the script is digitally signed, then the Help cannot be at the end of the script, because that location is reserved for the script's signature.

## Parameter Help

It's possible that an overzealous user of our **Test-Port** function may attempt to run it without first viewing the Help, which means they won't know that the **-Port** parameter is mandatory. This will cause the user to be prompted for a value.



```
PS C:\Scripts> Test-Port
cmdlet Test-Port at command pipeline position 1
Supply values for the following parameters:
Port:
```

Figure 19-7

If the user doesn't know what type of value the script is asking for when it prompts them with the **-Port** parameter, they'll either have to try to guess, which will generate an error if they guess wrong, or they could break out of the function by pressing **Ctrl+C**, and then go view the Help, by using **Get-Help**.

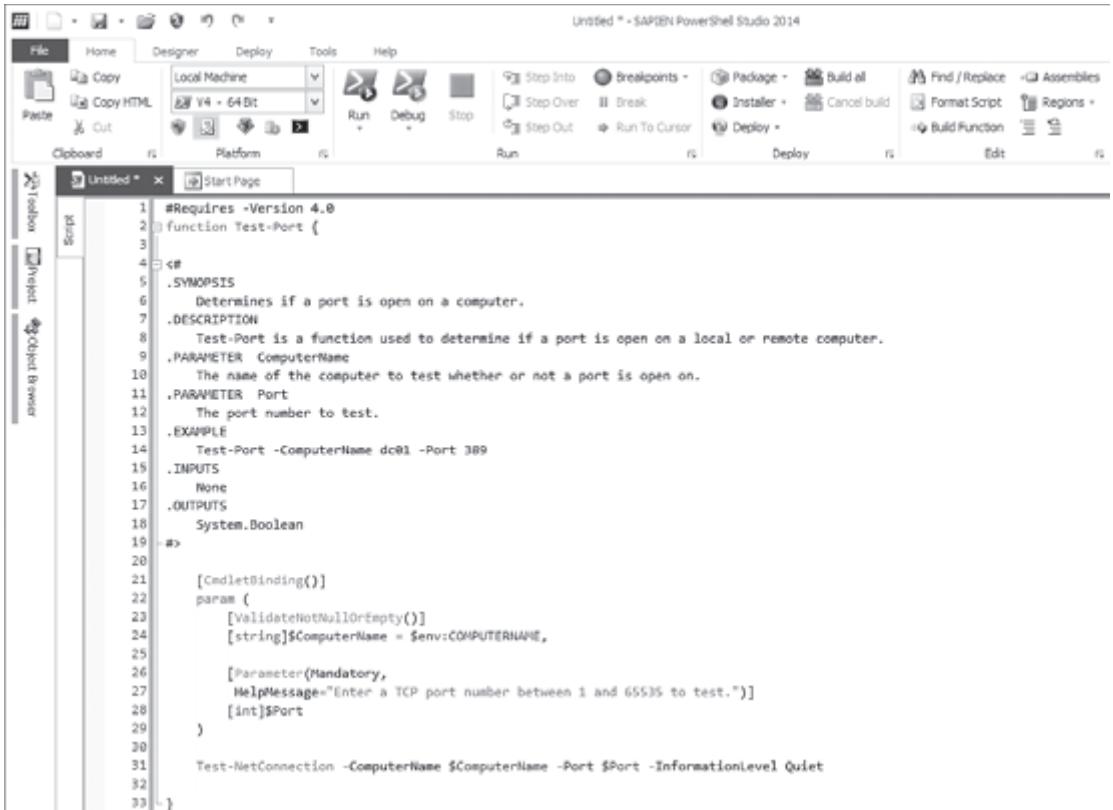
To provide help to the users who don't first use **Get-Help** to learn how to use our function, we'll add Help for the **-Port** parameter.

```
#Requires -Version 4.0
function Test-Port {

<#
.Synopsis
    Determines if a port is open on a computer.
.DESCRIPTION
    Test-Port is a function used to determine if a port is open on a local or remote computer.
.PARAMETER ComputerName
    The name of the computer to test whether or not a port is open on.
.PARAMETER Port
    The port number to test.
.EXAMPLE
    Test-Port -ComputerName dc01 -Port 389
.INPUTS
    None
.OUTPUTS
    System.Boolean
#>

[CmdletBinding()]
param (
    [ValidateNotNullOrEmpty()]
    [string]$ComputerName = $env:COMPUTERNAME,
    [Parameter(Mandatory,
        HelpMessage="Enter a TCP port number between 1 and 65535 to test.")]
    [int]$Port
)
Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
}
```

## Windows PowerShell: TFM



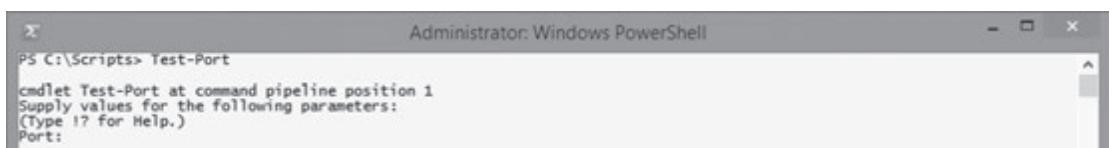
The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The title bar reads "Untitled \* - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Copy, Copy HTML, Cut, Run, Debug, Stop, Breakpoints, Package, Build all, Find / Replace, Assemblies, Format Script, Regions, and Build Function. The main window displays a PowerShell script titled "Test-Port.ps1". The script content is as follows:

```
1 #Requires -Version 4.0
2 function Test-Port {
3
4     <#
5         .SYNOPSIS
6             Determines if a port is open on a computer.
7         .DESCRIPTION
8             Test-Port is a function used to determine if a port is open on a local or remote computer.
9         .PARAMETER ComputerName
10            The name of the computer to test whether or not a port is open on.
11         .PARAMETER Port
12            The port number to test.
13         .EXAMPLE
14             Test-Port -ComputerName dc01 -Port 389
15         .INPUTS
16            None
17         .OUTPUTS
18            System.Boolean
19        #>
20
21        [CmdletBinding()]
22        param (
23            [ValidateNotNullOrEmpty()]
24            [string]$ComputerName = $env:COMPUTERNAME,
25
26            [Parameter(Mandatory,
27                HelpMessage="Enter a TCP port number between 1 and 65535 to test.")]
28            [int]$Port
29        )
30
31        Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
32    }
33 }
```

Figure 19-8

Now, when a user of our tool runs the function without specifying the mandatory **-Port** parameter, along with a valid value, they'll receive a slightly different message than before, and they can now type **!?** for Help.

Test-Port

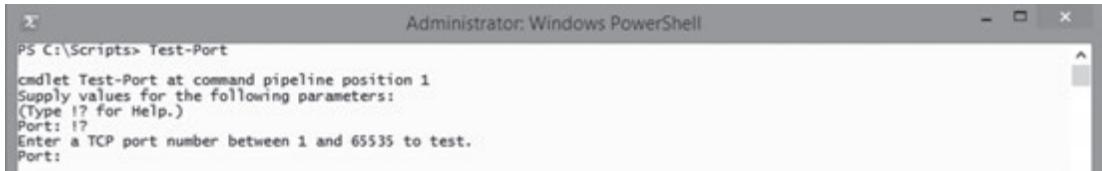


The screenshot shows an Administrator Windows PowerShell window. The title bar reads "Administrator: Windows PowerShell". The command entered is "PS C:\Scripts> Test-Port". The output shows the function's help message:

```
cmdlet Test-Port at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
Port:
```

Figure 19-9

Typing **!?** will display the following:



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\Scripts> Test-Port". The output displays the cmdlet's help information, including parameters and their descriptions. It asks for a port number between 1 and 65535 to test.

```

PS C:\Scripts> Test-Port
cmdlet Test-Port at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
Port: ?
Enter a TCP port number between 1 and 65535 to test.
Port:

```

Figure 19-10

## Inline documentation with Write- cmdlets

The final topic that we'll discuss is how to use the **Write-Verbose** and **Write-Debug** cmdlets.

### Write-Verbose

It's not uncommon to see single-line comments throughout a script. The problem with these types of comments is that no one will ever see them unless they open the script and read it. Our recommendation is to go ahead and add them to a **Write-Verbose** statement.

```

#Requires -Version 4.0
function Test-Port {

<#
.Synopsis
    Determines if a port is open on a computer.
.DESCRIPTION
    Test-Port is a function used to determine if a port is open on a local or remote computer.
.PARAMETER ComputerName
    The name of the computer to test whether or not a port is open on.
.PARAMETER Port
    The port number to test.
.EXAMPLE
    Test-Port -ComputerName dc01 -Port 389
.INPUTS
    None
.OUTPUTS
    System.Boolean
#>

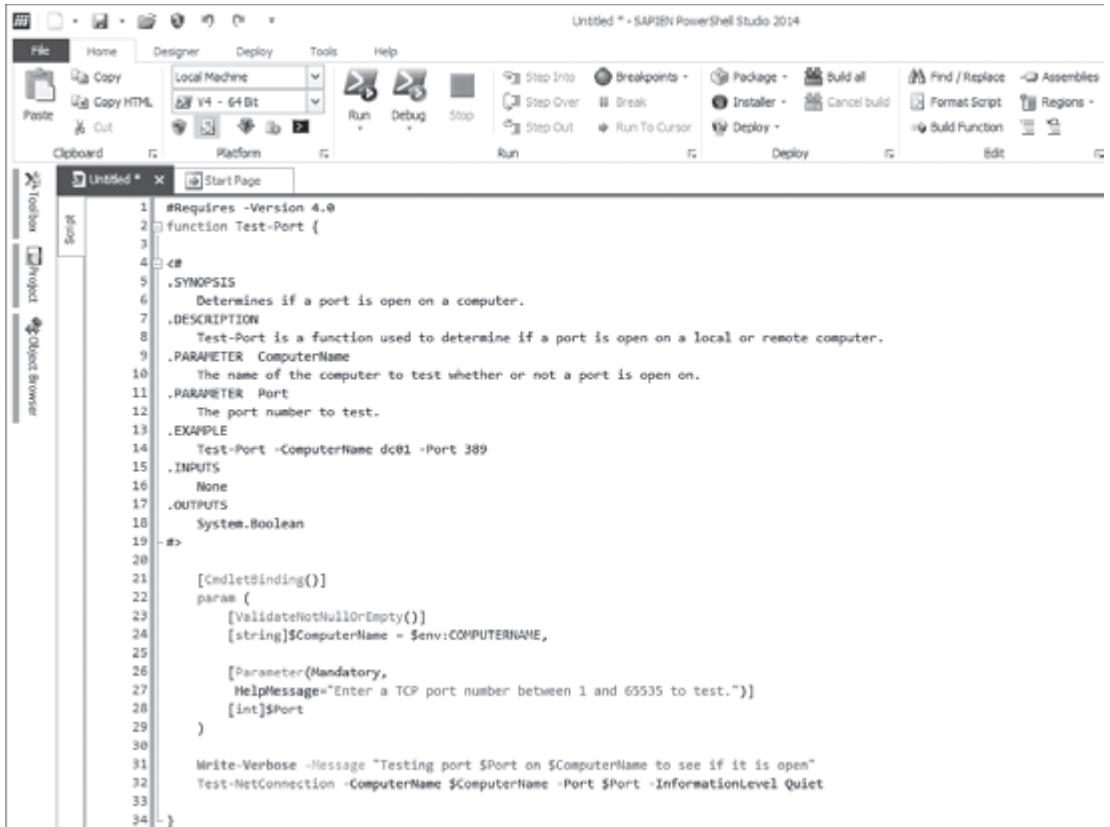
[CmdletBinding()]
param (
    [ValidateNotNullOrEmpty()]
    [string]$ComputerName = $env:COMPUTERNAME,

    [Parameter(Mandatory,
        HelpMessage="Enter a TCP port number between 1 and 65535 to test.")]
    [int]$Port
)

Write-Verbose "Testing port $Port on $ComputerName to see if it is open"
Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
}

```

## Windows PowerShell: TFM



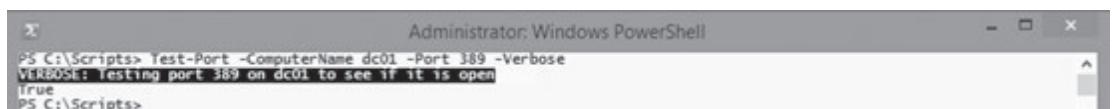
The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The main window displays a PowerShell script titled "Untitled". The script is a function named "Test-Port" with the following content:

```
1 #Requires -Version 4.0
2 Function Test-Port {
3
4     [CmdletBinding()}
5     param (
6         [ValidateNotNullOrEmpty()]
7         [string]$ComputerName = $env:COMPUTERNAME,
8
9         [Parameter(Mandatory,
10            HelpMessage="Enter a TCP port number between 1 and 65535 to test.")]
11        [int]$Port
12    )
13
14    Write-Verbose -Message "Testing port $Port on $ComputerName to see if it is open"
15    Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
16
17 }
```

Figure 19-11

Now, when you want to see the message that was added, you simply add the **-Verbose** parameter to the function.

```
Test-Port -ComputerName dc01 -Port 389 -Verbose
```



The screenshot shows an Administrator Windows PowerShell window. The command "Test-Port -ComputerName dc01 -Port 389 -Verbose" is run, and the output is:

```
PS C:\Scripts> Test-Port -ComputerName dc01 -Port 389 -Verbose
VERBOSE: Testing port 389 on dc01 to see if it is open
True
PS C:\Scripts>
```

Figure 19-12

## Write-Debug

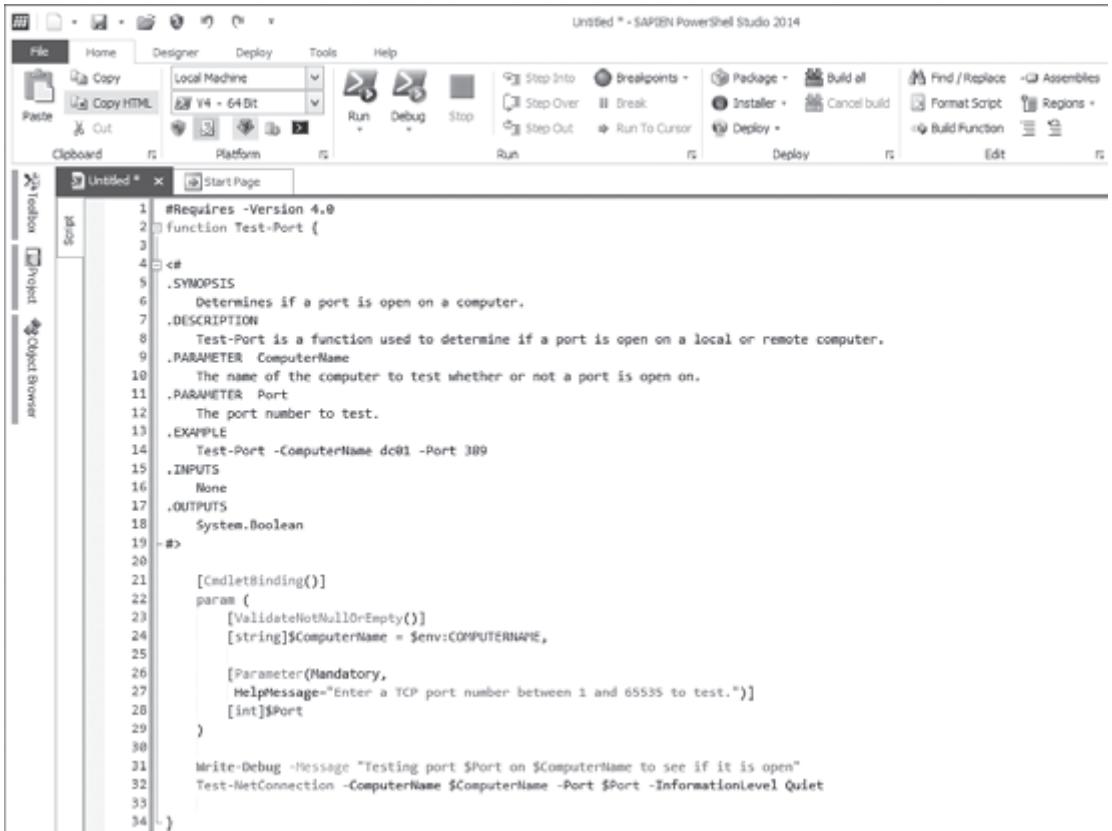
Maybe you have a script or function that's hundreds of lines long and you're attempting to troubleshoot it. Adding informational-related messages about what a script or function is supposed to be attempting to accomplish at specific intervals with the **Write-Debug** command can greatly assist your troubleshooting process.

```
#Requires -Version 4.0
function Test-Port {
    <#
    .SYNOPSIS
        Determines if a port is open on a computer.
    .DESCRIPTION
        Test-Port is a function used to determine if a port is open on a local or remote computer.
    .PARAMETER ComputerName
        The name of the computer to test whether or not a port is open on.
    .PARAMETER Port
        The port number to test.
    .EXAMPLE
        Test-Port -ComputerName dc01 -Port 389
    .INPUTS
        None
    .OUTPUTS
        System.Boolean
    #>

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string]$ComputerName = $env:COMPUTERNAME,

        [Parameter(Mandatory,
            HelpMessage="Enter a TCP port number between 1 and 65535 to test.")]
        [int]$Port
    )
    Write-Debug -Message "Testing port $Port on $ComputerName to see if it is open"
    Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
}
```

## Windows PowerShell: TFM



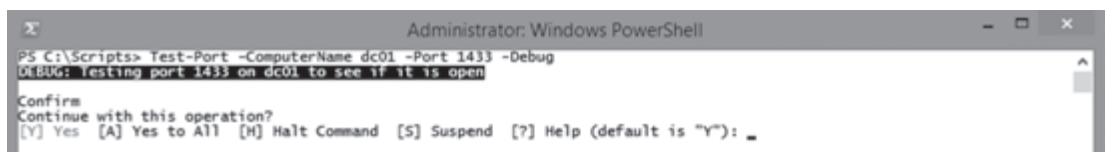
The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The main window displays a PowerShell script titled "Untitled". The script defines a function "Test-Port" with the following code:

```
1 #Requires -Version 4.0
2 function Test-Port {
3
4     <#
5         .SYNOPSIS
6             Determines if a port is open on a computer.
7         .DESCRIPTION
8             Test-Port is a function used to determine if a port is open on a local or remote computer.
9         .PARAMETER ComputerName
10            The name of the computer to test whether or not a port is open on.
11        .PARAMETER Port
12            The port number to test.
13        .EXAMPLE
14            Test-Port -ComputerName dc01 -Port 389
15        .INPUTS
16            None
17        .OUTPUTS
18            System.Boolean
19    >#
20
21    [CmdletBinding()]
22    param (
23        [ValidateNotNullOrEmpty()]
24        [String]$ComputerName = $env:COMPUTERNAME,
25
26        [Parameter(Mandatory,
27            HelpMessage="Enter a TCP port number between 1 and 65535 to test.")]
28        [Int]$Port
29    )
30
31    Write-Debug -Message "Testing port $Port on $ComputerName to see if it is open"
32    Test-NetConnection -ComputerName $ComputerName -Port $Port -InformationLevel Quiet
33
34 }
```

Figure 19-13

This time, we'll specify the **-Debug** parameter. The function will stop at the point where we defined our **Write-Debug** statement, and then prompt us for which action to take.

```
Test-Port -ComputerName dc01 -Port 1433 -Debug
```

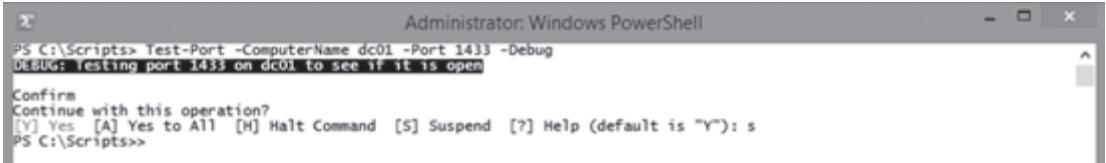


The screenshot shows an Administrator Windows PowerShell window. The command "Test-Port -ComputerName dc01 -Port 1433 -Debug" is run, and the output is:

```
PS C:\Scripts> Test-Port -ComputerName dc01 -Port 1433 -Debug
DEBUG: Testing port 1433 on dc01 to see if it is open
Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): -
```

Figure 19-14

Now, we're going to show you something really awesome. We're choosing option **S** to suspend the operation and return us to the PowerShell prompt.

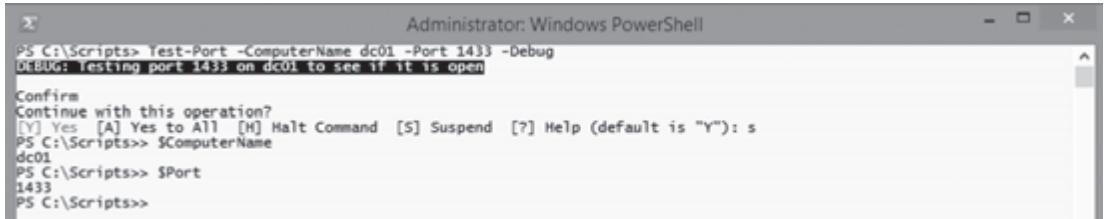


```
Administrator: Windows PowerShell
PS C:\Scripts> Test-Port -ComputerName dc01 -Port 1433 -Debug
DEBUG: Testing port 1433 on dc01 to see if it is open

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): s
PS C:\Scripts>>
```

Figure 19-15

You may not have noticed in Figure 19-15, but we're now at a special nested prompt, designated by the two greater than signs >> at the end of the prompt. At this point, we can check the values of our variables by simply entering a dollar sign, followed by the variable name, and then pressing **Enter**.



```
Administrator: Windows PowerShell
PS C:\Scripts> Test-Port -ComputerName dc01 -Port 1433 -Debug
DEBUG: Testing port 1433 on dc01 to see if it is open

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): s
PS C:\Scripts>> $ComputerName
dc01
PS C:\Scripts>> $Port
1433
PS C:\Scripts>>
```

Figure 19-16

If that alone wasn't awesome enough, we can now actually change the values of our variables, and then have the function continue with the new values.



```
Administrator: Windows PowerShell
PS C:\Scripts> Test-Port -ComputerName dc01 -Port 1433 -Debug
DEBUG: Testing port 1433 on dc01 to see if it is open

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): s
PS C:\Scripts>> $ComputerName
dc01
PS C:\Scripts>> $Port
1433
PS C:\Scripts>> $ComputerName = 'sql01'
PS C:\Scripts>> exit

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): y
WARNING: Ping to sql01 failed -- Status: TimedOut
True
PS C:\Scripts>
```

Figure 19-17

In Figure 19-17, we changed the value of our **ComputerName** variable, typed **exit** to exit from the debug suspended mode, and then typed **Y** to confirm continuing with the operation.

## Exercise 19 – Adding comment-based Help

Your company recently had a significant amount of turnover and there are several new junior-level admins who don't understand how to use the tool that you created in Chapter 17, so you've been asked to document that tool.

### Task 1

Add comment-based Help to the function that you created in the exercise at the end of Chapter 17.

### Task 2

Add the necessary code to have the function list additional information when the **-Verbose** parameter is specified when the function is called.

## Chapter 20

# Debugging and error handling

### The purpose of a quality debugger

Debugging in PowerShell is the process of finding and reducing the number of issues or potential issues that could cause syntax or logic errors in a script, function, workflow, or module. The purpose of a quality debugger is to assist you in locating and eliminating those issues.

### What could go wrong?

Design for defense. We've always heard that the best offense is a good defense and the same holds true when designing your PowerShell code. We're IT professionals, not developers, so don't let the word "code" scare you away from using PowerShell. You don't have to be a developer to be extremely effective with PowerShell. In fact, PowerShell is designed especially for system administration. Don't believe us? Take a look at the first paragraph on PowerShell from Microsoft TechNet.

# Scripting with Windows PowerShell

140 out of 207 rated this helpful - Rate this topic

Updated: September 3, 2013

Applies To: Windows PowerShell 2.0, Windows PowerShell 3.0, Windows PowerShell 4.0

Windows PowerShell® is a task-based command-line shell and scripting language designed especially for system administration. Built on the .NET Framework, Windows PowerShell helps IT professionals and power users control and automate the administration of the Windows operating system and applications that run on Windows.

Figure 20-1

Source: <http://tinyurl.com/nfmnhm>

We use the word “commands” as a generic term to refer to PowerShell scripts, functions, workflows, and modules, so if you see that word you’ll know what we’re talking about.

What do we mean by design for defense? When designing your PowerShell commands, think about what could go wrong if you handed your command over to someone else who doesn’t know anything about it—someone like a junior-level admin or help desk worker.

## PowerShell debugger

The debugger in PowerShell version 4 has been updated to debug PowerShell commands that are running locally, as well as commands that are running on remote machines.

There are a number of cmdlets in PowerShell that can be used for debugging.

```
Get-Command -Noun PSBreakPoint, PSCallStack
```

CommandType	Name	ModuleName
Cmdlet	Disable-PSBreakpoint	Microsoft.PowerShell.Utility
Cmdlet	Enable-PSBreakpoint	Microsoft.PowerShell.Utility
Cmdlet	Get-PSBreakpoint	Microsoft.PowerShell.Utility
Cmdlet	Get-PSCallStack	Microsoft.PowerShell.Utility
Cmdlet	Remove-PSBreakpoint	Microsoft.PowerShell.Utility
Cmdlet	Set-PSBreakpoint	Microsoft.PowerShell.Utility

Figure 20-2

## Setting BreakPoints

We've purposely broken the **Get-SystemInventory** function that we created in a previous chapter. We'll first dot source our script that contains our function to load it into memory, so that we can access it.

```
. .\Get-SystemInventory.ps1
```

Now, we're going to set a breakpoint by using the **Set-PSBreakPoint** cmdlet. This will set a breakpoint at the very beginning of our function.

```
Set-PSBreakPoint -Command Get-SystemInventory
```

ID	Script	Line	Command	Variable	Action
0			Get-SystemInventory		

Figure 20-3

Now that we've set a breakpoint, we call our **Get-SystemInventory** function and we're given information on how to receive Help on using debug mode. Our prompt also changes and is preceded by [DBG] to designate that we're in debug mode.

```
Get-SystemInventory
```

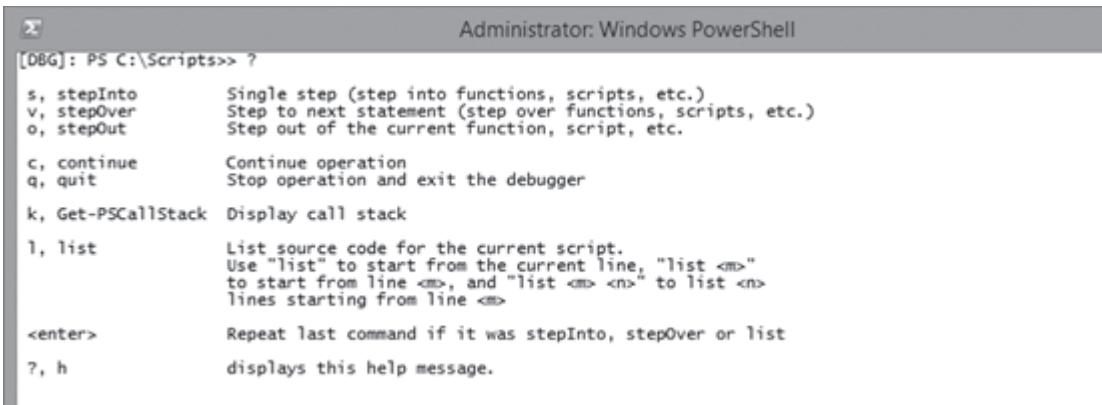
```
PS C:\Scripts> Get-SystemInventory
Entering debug mode. Use h or ? for help.
Hit Command breakpoint on 'Get-SystemInventory'
At C:\Scripts\Get-SystemInventory.ps1:1 char:30
+ function Get-SystemInventory {
+     ~~~~~
[DBG]: PS C:\Scripts>> _
```

Figure 20-4

If you're unsure about the available options for debug mode, press the question mark or the letter "H", as shown in Figure 20-5.

?

## Windows PowerShell: TFM



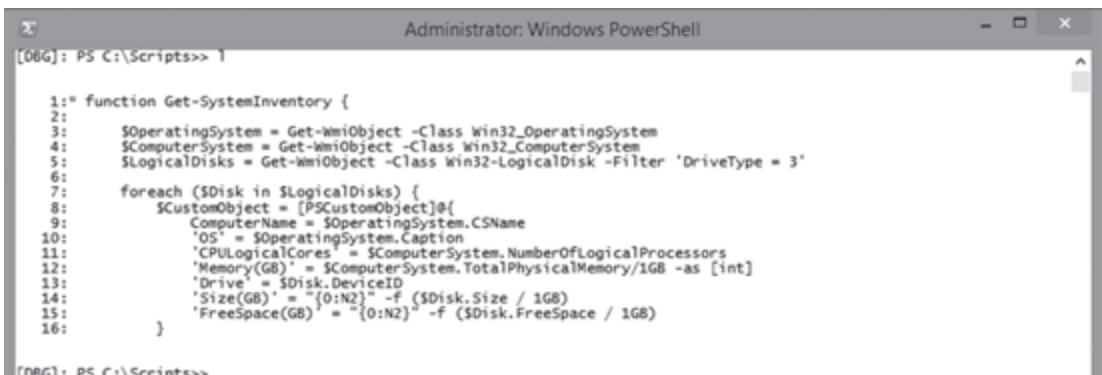
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "[DBG]: PS C:\Scripts>> ?". The output displays a list of debugger commands:

- s, stepInto Single step (step into functions, scripts, etc.)
- v, stepOver Step to next statement (step over functions, scripts, etc.)
- o, stepOut Step out of the current function, script, etc.
- c, continue Continue operation
- q, quit Stop operation and exit the debugger
- k, Get-PSCallStack Display call stack
- l, list List source code for the current script.  
Use "list" to start from the current line, "list <m>"  
to start from line <m>, and "list <m> <n>" to list <n>  
lines starting from line <m>
- <enter> Repeat last command if it was stepInto, stepOver or list
- ?, h Displays this help message.

Figure 20-5

Entering the letter "L" lists the source code for our function and we can see the breakpoint was placed at the very beginning of our function.

L



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "[DBG]: PS C:\Scripts>> l". The output displays the source code of a PowerShell function:

```
1:# function Get-SystemInventory {
2:
3:     $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
4:     $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
5:     $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'
6:
7:     foreach ($Disk in $LogicalDisks) {
8:         $CustomObject = [PSCustomObject]@{
9:             ComputerName = $OperatingSystem.CSName
10:            'OS' = $OperatingSystem.Caption
11:            'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
12:            'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/1GB -as [int]
13:            'Drive' = $Disk.DeviceID
14:            'Size(GB)' = '{0:N2}' -f ($Disk.Size / 1GB)
15:            'FreeSpace(GB)' = '{0:N2}' -f ($Disk.FreeSpace / 1GB)
16:        }
17:    }
```

Figure 20-6

Now, we'll use the "S" key to step through each line of our function. Once an error is generated, we'll know the problem exists with the previous command, the one that ran just before the error message. It's also possible that the error itself isn't in the command on that line, but something that it depends on.

S

```
[DBG]: PS C:\Scripts>> s
At C:\Scripts\Get-SystemInventory.ps1:3 char:5
+ $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
+ ~~~~~
[DBG]: PS C:\Scripts>> s
At C:\Scripts\Get-SystemInventory.ps1:4 char:5
+ $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem
+ ~~~~~
[DBG]: PS C:\Scripts>> s
At C:\Scripts\Get-SystemInventory.ps1:5 char:5
+ $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = ...'
+ ~~~~~
[DBG]: PS C:\Scripts>> s
Get-WmiObject : Invalid query 'select * from Win32_LogicalDisk where DriveType = 3'
At C:\Scripts\Get-SystemInventory.ps1:5 char:21
+ $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = ...'
+ ~~~~~
+ CategoryInfo          : InvalidArgument: (:) [Get-WmiObject], ManagementException
+ FullyQualifiedErrorId : GetWMIManagementException,Microsoft.PowerShell.Commands.GetWmiObjectCommand
At C:\Scripts\Get-SystemInventory.ps1:7 char:23
+ foreach ($Disk in $LogicalDisks) {
+ ~~~~~
[DBG]: PS C:\Scripts>> q
PS C:\Scripts>
```

Figure 20-7

Even though we've used the "Q" key to quit debug mode, as shown in Figure 20-7, the breakpoint is still set and we'd be right back in debug mode if we called the **Get-SystemInventory** function again.

Use the **Remove-PSBreakPoint** cmdlet to remove a breakpoint.

```
Get-PSBreakPoint -Command Get-SystemInventory | Remove-PSBreakPoint
```

```
Administrator: Windows PowerShell
PS C:\Scripts> Get-PSBreakpoint -Command Get-SystemInventory | Remove-PSBreakpoint
PS C:\Scripts> PS C:\Scripts>
```

Figure 20-8

It's also possible to disable breakpoints temporarily by using the **Disable-PSBreakPoint** cmdlet. You can re-enable them later by using the **Enable-PSBreakPoint** cmdlet.

## About \$ErrorActionPreference

**\$ErrorActionPreference** is a built-in preference variable that determines how PowerShell treats non-terminating errors. By default, it's set to **Continue**, which causes an error to be displayed and then for the command to continue processing. Remember, this setting only affects non-terminating errors.

We consider it a best practice to leave this variable set to its default setting. Don't change it unless you have a really good reason for doing so and only when no alternatives exist. With that said, we consider best practices to be a starting point and if you're going to change the value of this variable, we recommend changing it back to the default setting of **Continue** immediately after completing the task that required it to be changed.

## Generic error handling

Error handling is a method of adding logic to your commands that specifies what action to take if (or when) an error occurs.

## Using -ErrorAction and -ErrorVariable

**-ErrorAction** and **-ErrorVariable** are two of the common parameters that are available to every cmdlet. They're also available to any command that you create that has the [**CmdletBinding()**] or **Param** attribute defined, as well as to all workflows that you create.

The **-ErrorAction** parameter is used to override the **\$ErrorActionPreference** setting on a per command basis. This parameter has no effect on terminating errors. A terminating error is an error that prevents a command from completing successfully.

First, to demonstrate the default behavior when using the default setting of **Continue** for the **\$ErrorActionPreference** variable without the **-ErrorAction** parameter being specified, we'll attempt to copy a file that doesn't exist.

```
Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp'
Write-Output "Continued processing after the error"
```



Figure 20-9

We run the script, which generates a non-terminating error and it continues to the second line. Since the **-ErrorAction** parameter was not specified, it uses the setting of **\$ErrorActionPreference**, which defaults to **Continue**.

```
.\NonTerminatingError.ps1
```

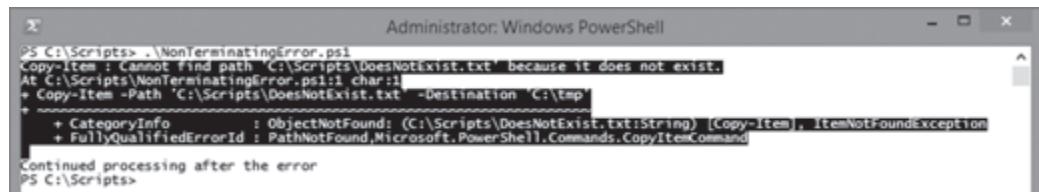


Figure 20-10

## Debugging and error handling

We've created another script similar to our previous one, except that we've specified the **-ErrorAction** parameter, with a value of **Stop**.

```
Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp' -ErrorAction Stop  
Write-Output "Continued processing after the error"
```



Figure 20-11

When this script runs, it runs the first command, which generates an error, and then it stops because the **-ErrorAction** parameter is specified with a value of **Stop**. This turns our non-terminating error into a terminating error. Notice that the second line in the script was not executed this time.

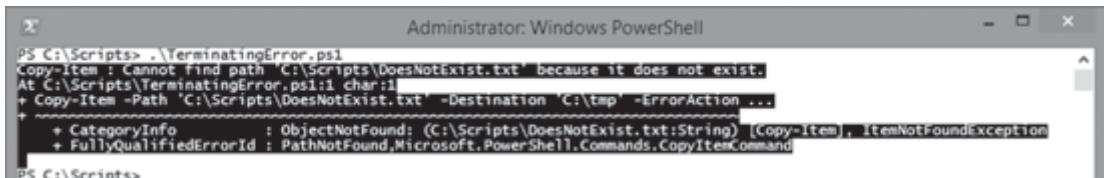
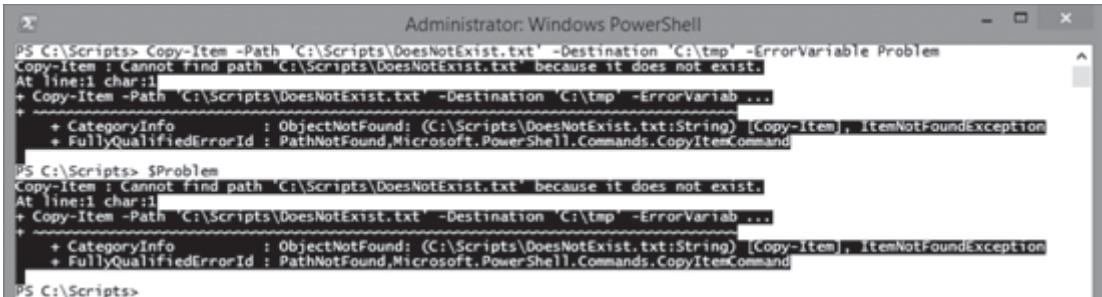


Figure 20-12

It's very important that you understand how to turn a non-terminating error into a terminating one, so make sure that you understand this concept before continuing.

The **-ErrorVariable** parameter is used to store the error message that's generated by a command. Notice that the name of the variable itself is specified without a \$.

```
Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp' -ErrorVariable Problem  
$Problem
```



```
Administrator: Windows PowerShell
PS C:\Scripts> Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp' -ErrorVariable Problem
Copy-Item : Cannot find path 'C:\Scripts\DoesNotExist.txt' because it does not exist.
At line:1 char:1
+ Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp' -ErrorVariab ...
+ CategoryInfo          : ObjectNotFound: (C:\Scripts\DoesNotExist.txt:String) [Copy-Item], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.CopyItemCommand

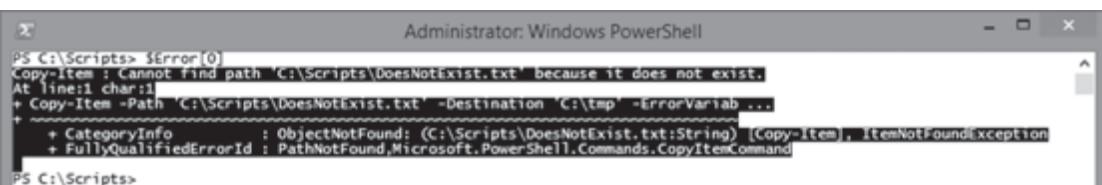
PS C:\Scripts> $Problem
Copy-Item : Cannot find path 'C:\Scripts\DoesNotExist.txt' because it does not exist.
At line:1 char:1
+ Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp' -ErrorVariab ...
+ CategoryInfo          : ObjectNotFound: (C:\Scripts\DoesNotExist.txt:String) [Copy-Item], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.CopyItemCommand

PS C:\Scripts>
```

Figure 20-13

We're sure you've already learned that in PowerShell there are many different ways to accomplish the same task. **\$Error** is an automatic variable that exists in PowerShell without you having to do anything. An additional way to view the most recent error message is to return the first element of the error array.

```
$Error[0]
```



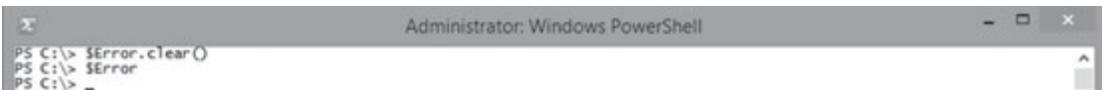
```
Administrator: Windows PowerShell
PS C:\Scripts> $Error[0]
Copy-Item : Cannot find path 'C:\Scripts\DoesNotExist.txt' because it does not exist.
At line:1 char:1
+ Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp' -ErrorVariab ...
+ CategoryInfo          : ObjectNotFound: (C:\Scripts\DoesNotExist.txt:String) [Copy-Item], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.CopyItemCommand

PS C:\Scripts>
```

Figure 20-14

One of the easiest ways to capture all of the errors from a particular command is to first clear all of the errors in the **\$Error** array, by using the **.clear()** method.

```
$Error.clear()
$Error
```



```
Administrator: Windows PowerShell
PS C:\> $Error.clear()
PS C:\> $Error
PS C:\> =
```

Figure 20-15

Then run the command or commands that you want to capture the errors from. Notice that we ran the second command in Figure 20-16 three times, using the **Range** operator to generate three errors with it, along with one error from the first command that we ran.

```
.\TerminatingError.ps1
1..3 | Copy-Item -Path 'C:\Scripts\a.txt' -Destination 'C:\tmp'
```

```
PS C:\Scripts> .\TerminatingError.ps1
Copy-Item : Cannot find path 'C:\Scripts\DoesNotExist.txt' because it does not exist.
At C:\Scripts\TerminatingError.ps1:1 char:1
+ Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp' -ErrorAction ...
+ CategoryInfo          : ObjectNotFound: (C:\Scripts\DoesNotExist.txt:String) [Copy-Item], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.CopyItemCommand

PS C:\Scripts> 1..3 | Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp'
Copy-Item : The input object cannot be bound to any parameters for the command either because the command does not
take pipeline input or the input and its properties do not match any of the parameters that take pipeline input.
At Line:1 char:8
+ 1..3 | Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp'
+ CategoryInfo          : InvalidArgument: (1:Int32) [Copy-Item], ParameterBindingException
+ FullyQualifiedErrorId : InputObjectNotBound,Microsoft.PowerShell.Commands.CopyItemCommand

Copy-Item : The input object cannot be bound to any parameters for the command either because the command does not
take pipeline input or the input and its properties do not match any of the parameters that take pipeline input.
At Line:1 char:8
+ 1..3 | Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp'
+ CategoryInfo          : InvalidArgument: (2:Int32) [Copy-Item], ParameterBindingException
+ FullyQualifiedErrorId : InputObjectNotBound,Microsoft.PowerShell.Commands.CopyItemCommand

Copy-Item : The input object cannot be bound to any parameters for the command either because the command does not
take pipeline input or the input and its properties do not match any of the parameters that take pipeline input.
At Line:1 char:8
+ 1..3 | Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp'
+ CategoryInfo          : InvalidArgument: (3:Int32) [Copy-Item], ParameterBindingException
+ FullyQualifiedErrorId : InputObjectNotBound,Microsoft.PowerShell.Commands.CopyItemCommand

PS C:\Scripts> _
```

Figure 20-16

See the *Special Operators* section of the `about_operators` Help topic in PowerShell to learn more about the **Range** operator.

We've often found that the initial error that is produced by a command generates a cascading effect and causes all of the errors beyond that point. Resolving this initial error will usually either solve the problem altogether or allow the command to process further until another problem is reached. Since we cleared the `$Error` array before we ran our commands, we can view the first error that occurred by specifying `-1` for the element number, without having to know how many errors occurred.

```
$Error[-1]
```

```
PS C:\Scripts> $Error[-1]
Copy-Item : Cannot find path 'C:\Scripts\DoesNotExist.txt' because it does not exist.
At C:\Scripts\TerminatingError.ps1:1 char:1
+ Copy-Item -Path 'C:\Scripts\DoesNotExist.txt' -Destination 'C:\tmp' -ErrorAction ...
+ CategoryInfo          : ObjectNotFound: (C:\Scripts\DoesNotExist.txt:String) [Copy-Item], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.CopyItemCommand

PS C:\Scripts> _
```

Figure 20-17

By default, the **\$Error** array only stores the last 256 errors. The **\$MaximumErrorCount** preference variable controls the number of errors that are kept, just in case you ever need to change its value.

```
$MaximumErrorCount
```

```
Administrator: Windows PowerShell
PS C:\Scripts> $MaximumErrorCount
256
PS C:\Scripts>
```

Figure 20-18

Changing the value of the **\$MaximumErrorCount** preference variable only changes the number of errors that will be kept for that session of PowerShell. Once PowerShell is closed and reopened, the number of errors will revert to 256. If you need to change this preference variable permanently, add the code to change it to your PowerShell profile.

```
$MaximumErrorCount = 500
$MaximumErrorCount
```

```
Administrator: Windows PowerShell
PS C:\Scripts> $MaximumErrorCount = 500
PS C:\Scripts> $MaximumErrorCount
500
PS C:\Scripts>
```

Figure 20-19

Keep in mind that if the **-ErrorAction** parameter is specified with a value of **Ignore** when an error is generated while running a command, the error will not be displayed on the screen, it will not be added to the variable specified via the **-ErrorVariable** parameter, and it will not be added to the **\$Error** automatic variable.

## Inline debugging with Write-Debug and Write-Verbose

The **Write-Debug** cmdlet is used to perform inline debugging. We've added three **Write-Debug** statements to our **Get-SystemInventory** function.

```
function Get-SystemInventory {
    [CmdletBinding()]
    param()

    Write-Debug -Message "Attempting to Query Win32_OperatingSystem"
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem
```

## Debugging and error handling

```
Write-Debug -Message "Attempting to Query Win32_ComputerSystem"
$ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem

Write-Debug -Message "Attempting to Query Win32_LogicalDisk"
$LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

foreach ($Disk in $LogicalDisks) {
    $CustomObject = [PSCustomObject]@{
        ComputerName = $OperatingSystem.CSName
        'OS' = $OperatingSystem.Caption
        'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
        'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/1GB -as [int]
        'Drive' = $Disk.DeviceID
        'Size(GB)' = "{0:N2}" -f ($Disk.Size / 1GB)
        'FreeSpace(GB)' = "{0:N2}" -f ($Disk.FreeSpace / 1GB)
    }

    $CustomObject.PSTypeNames.Insert(0,'TFM.SystemInventory')

    Write-Output $CustomObject
}

}
```

The screenshot shows the SAP2N PowerShell Studio 2014 interface. The title bar reads 'Get-SystemInventory.ps1 - SAP2N PowerShell Studio 2014'. The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Copy, Copy HTML, Cut, Run, Debug, Stop, Step Into, Breakpoints, Package, Installer, Find/Replace, Assemblies, Format Script, Regions, Build Function, and Build All. The main window displays the PowerShell script 'Get-SystemInventory.ps1'. The script code is as follows:

```
function Get-SystemInventory {
    [CmdletBinding()]
    param()

    Write-Debug -Message "Attempting to Query Win32_OperatingSystem"
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem

    Write-Debug -Message "Attempting to Query Win32_ComputerSystem"
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem

    Write-Debug -Message "Attempting to Query Win32_LogicalDisk"
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    foreach ($Disk in $LogicalDisks) {
        $CustomObject = [PSCustomObject]@{
            ComputerName = $OperatingSystem.CSName
            'OS' = $OperatingSystem.Caption
            'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
            'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/1GB -as [int]
            'Drive' = $Disk.DeviceID
            'Size(GB)' = "{0:N2}" -f ($Disk.Size / 1GB)
            'FreeSpace(GB)' = "{0:N2}" -f ($Disk.FreeSpace / 1GB)
        }

        $CustomObject.PSTypeNames.Insert(0,'TFM.SystemInventory')

        Write-Output $CustomObject
    }
}
```

Figure 20-20

There's an issue with our **Get-SystemInventory** function that causes it to generate an error when we attempt to run it. To assist us in locating the error, we can specify the **-Debug** parameter when calling the function, which tells it to pause and ask how to continue at each of the **Write-Debug** statements.

To locate our error, we'll first dot source the script that contains our **Get-SystemInventory** function, and then we'll specify **Y** to continue. We'll then press **S** for suspend, which returns us to a prompt, but notice that it's a nested prompt. This is where we can check the values of our variables and even change their values if we so desire. Typing **Exit** returns us to running our command from where we left off before we entered the nested prompt.

```
. .\Get-SystemInventory.ps1
Get-SystemInventory.ps1
Y
S
Exit
```

The screenshot shows an Administrator Windows PowerShell window. The command entered was ". .\Get-SystemInventory.ps1" followed by "Get-SystemInventory -Debug". The output includes several "DEBUG: Attempting to Query Win32\_OperatingSystem" messages and two nested "Confirm Continue with this operation?" prompts. The first prompt has "[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): y" and the second has "[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): s". Below these, the script outputs system information like SystemDirectory, Organization, BuildNumber, RegisteredUser, SerialNumber, and Version. Finally, it exits with "PS C:\Scripts>> exit" and another confirmation prompt: "[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): \_".

Figure 20-21

You can press the default of **Enter** or **Y** to continue to the next **Write-Debug** statement.

```
Exit
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered was "PS C:\Scripts>> exit". A "Confirm" dialog box appears, asking "Continue with this operation? [Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y")". Below the dialog, an error message is displayed: "DEBUG: Attempting to Query Win32\_LogicalDisk". Another "Confirm" dialog follows with the same options. The error message continues: "Get-WmiObject : Invalid query 'Select \* From Win32\_LogicalDisk where DriveType = 3' At C:\Scripts\Get-SystemInventory.ps1:13 char:21 + \$LogicalDisks = Get-WmiObject -Class Win32\_LogicalDisk -Filter 'DriveType = ... + CategoryInfo : InvalidArgument: (:) [Get-WmiObject], ManagementException + FullyQualifiedErrorId : GetWMIManagementException,Microsoft.PowerShell.Commands.GetWmiObjectCommand". The command "PS C:\Scripts> " is shown at the bottom.

Figure 20-22

We can see that our **Get-SystemInventory** function errors out after attempting to query information from the **Win32\_LogicalDisk** WMI class, so we know the issue is with that particular portion of the function. By looking closely at the error message, we can see that there is a syntax error, because the WMI class is misspelled as “Win32-LogicalDisk” and it should be “Win32\_LogicalDisk”.

We’ve now changed our function from using the **Write-Debug** cmdlet to using **Write-Verbose**. **Write-Verbose** is something that we use in almost every command that we write and that is longer than a few lines. It allows us to receive the verbose output by specifying the **-Verbose** parameter when calling the command.

```
function Get-SystemInventory {
    [CmdletBinding()]
    param()

    Write-Verbose -Message "Attempting to Query Win32_OperatingSystem"
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem

    Write-Verbose -Message "Attempting to Query Win32_ComputerSystem"
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem

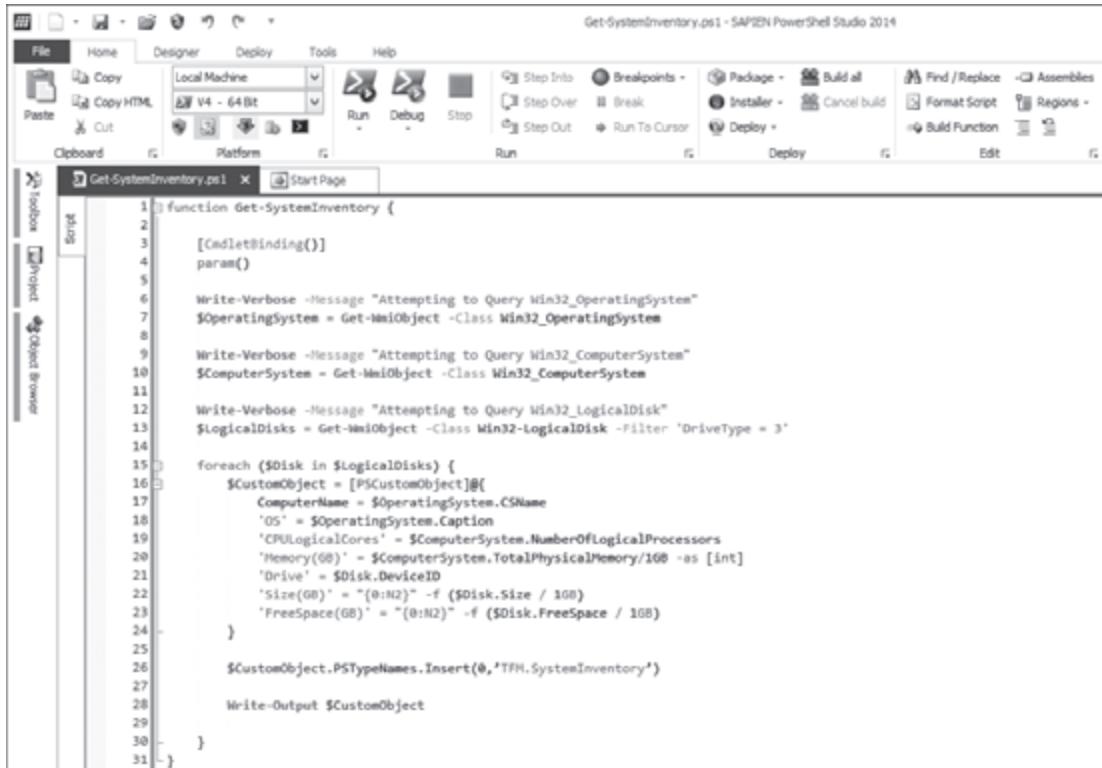
    Write-Verbose -Message "Attempting to Query Win32_LogicalDisk"
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    foreach ($Disk in $LogicalDisks) {
        $CustomObject = [PSCustomObject]@{
            ComputerName = $OperatingSystem.CSName
            'OS' = $OperatingSystem.Caption
            'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
            'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/1GB -as [int]
            'Drive' = $Disk.DeviceID
            'Size(GB)' = "{0:N2}" -f ($Disk.Size / 1GB)
            'FreeSpace(GB)' = "{0:N2}" -f ($Disk.FreeSpace / 1GB)
        }

        $CustomObject.PSTypeNames.Insert(0,'TFM.SystemInventory')

        Write-Output $CustomObject
    }
}
```

## Windows PowerShell: TFM



The screenshot shows the SAPiEN PowerShell Studio interface. The title bar reads "Get-SystemInventory.ps1 - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, Help. The toolbar has icons for Copy, Copy HTML, Cut, Run, Debug, Stop, Step Into, Breakpoints, Package, Installer, Find / Replace, Assemblies, Format Script, Regions, and Build Function. Below the toolbar are tabs for Clipboard, Platform, Run, Deploy, and Edit. The main pane displays the PowerShell script "Get-SystemInventory.ps1". The script uses Write-Verbose cmdlets to log its actions and retrieves system information using Get-WmiObject cmdlets for Win32\_OperatingSystem, Win32\_ComputerSystem, and Win32\_LogicalDisk classes.

```
function Get-SystemInventory {
    [CmdletBinding()]
    param()

    Write-Verbose -Message "Attempting to Query Win32_OperatingSystem"
    $OperatingSystem = Get-WmiObject -Class Win32_OperatingSystem

    Write-Verbose -Message "Attempting to Query Win32_ComputerSystem"
    $ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem

    Write-Verbose -Message "Attempting to Query Win32_LogicalDisk"
    $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = 3'

    foreach ($Disk in $LogicalDisks) {
        $CustomObject = [PSCustomObject]@{
            ComputerName = $OperatingSystem.CSName
            'OS' = $OperatingSystem.Caption
            'CPULogicalCores' = $ComputerSystem.NumberOfLogicalProcessors
            'Memory(GB)' = $ComputerSystem.TotalPhysicalMemory/100 -as [int]
            'Drive' = $Disk.DeviceID
            'Size(GB)' = "{0:N2}" -f ($Disk.Size / 100)
            'FreeSpace(GB)' = "{0:N2}" -f ($Disk.FreeSpace / 100)
        }

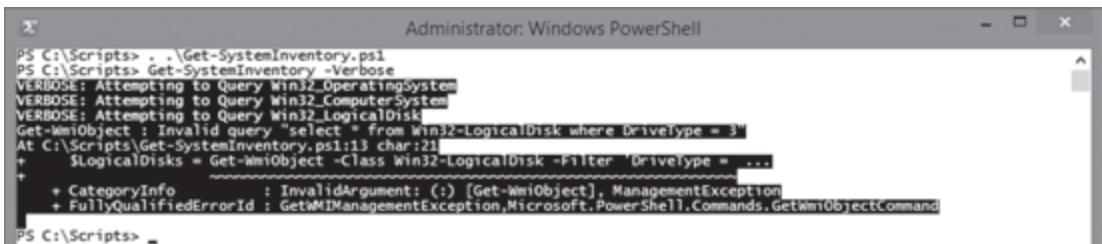
        $CustomObject.PSTypeNames.Insert(0,'TFM.SystemInventory')
    }

    Write-Output $CustomObject
}
```

Figure 20-23

As you can see, we receive similar results when using the **Write-Verbose** cmdlet, except there's no way to stop the script and check values of a particular variable.

```
. .\Get-SystemInventory.ps1
Get-SystemInventory -Verbose
```



The screenshot shows an Administrator Windows PowerShell window. The command ". .\Get-SystemInventory.ps1" is run, followed by "Get-SystemInventory -Verbose". The output shows verbose messages indicating the script is attempting to query Win32\_OperatingSystem, Win32\_ComputerSystem, and Win32\_LogicalDisk. It then logs an error message about an invalid query for logical disks and provides detailed error information including CategoryInfo and FullyQualifiedErrorId.

```
PS C:\Scripts> . .\Get-SystemInventory.ps1
PS C:\Scripts> Get-SystemInventory -Verbose
VERBOSE: Attempting to Query Win32_OperatingSystem
VERBOSE: Attempting to Query Win32_ComputerSystem
VERBOSE: Attempting to Query Win32_LogicalDisk
Get-WmiObject : Invalid query 'select * from Win32_LogicalDisk where DriveType = 3'
At C:\Scripts\Get-SystemInventory.ps1:13 char:21
+ $LogicalDisks = Get-WmiObject -Class Win32_LogicalDisk -Filter 'DriveType = ...
+                                     ~~~~~
+ CategoryInfo          : InvalidArgument: (:) [Get-WmiObject], ManagementException
+ FullyQualifiedErrorId : GetWMIManagementException,Microsoft.PowerShell.Commands.GetWmiObjectCommand
PS C:\Scripts>
```

Figure 20-24

## Handling error output

There are a couple of different ways of handling error output in PowerShell. Trapping errors through the use of the **Trap** keyword is a method of handling errors that was introduced in PowerShell version 1. PowerShell version 2 introduced a new way of handling errors though the use of “Try-Catch-Finally”.

We’ll be focusing on Try-Catch-Finally in this chapter, but if you’re interested in learning more about the **Trap** keyword, see the about\_Trap Help topic in PowerShell.

## Using Try-Catch-Finally

Try-Catch-Finally is a block of code that is used to handle terminating errors. Remember when we showed you earlier how to turn a non-terminating error into a terminating error? The reason it was so important to learn that concept is because Try-Catch-Finally only handles terminating errors. So if you want an error to be handled, you must turn it into a terminating error (if it’s not already one).

If a terminating error occurs from the code that’s specified in the **Try** block, the code specified in the **Catch** block will execute. If you’re going to use Try-Catch-Finally, **Try** is required and either **Catch** or **Finally** is also required, although all three can be specified. This may be a little different than what you’ve read online, so we’ll go ahead and attempt to dot source a script that we’ve added a **Try** block to, but not a **Catch** or **Finally** block, so you can read the error message for yourself.

```
. .\Get-LocalUser.ps1
```

```
Administrator: Windows PowerShell
PS C:\Scripts> . .\Get-LocalUser.ps1
At C:\Scripts\Get-LocalUser.ps1:23 char:10
+         }
+         ~~~~~
The Try statement is missing its Catch or Finally block.
+ CategoryInfo          : ParserError: (:) [ParseException]
+ FullyQualifiedErrorId : MissingCatchOrFinally
PS C:\Scripts> _
```

Figure 20-25

**Finally** is the one that we typically see omitted, as it executes regardless, whether an error occurs or not. We normally see it used for cleanup, when it is specified.

We’re now going to re-use the **Get-LocalUser** PowerShell function that we created in a previous chapter. We’ve added a **Try** block and a **Catch** block surrounding the command where we think that an error could occur.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1,9)]
        [int]$SIDType = 1
    )
}
```

```

BEGIN {
    $Params = @{
        ClassName = 'Win32_Account'
        Filter = "SIDType=$SIDType and LocalAccount=$true"
        ErrorAction = 'Stop'
    }
}

PROCESS {
    try {
        Get-CimInstance @Params -ComputerName $ComputerName
    }
    catch {
        Write-Warning -Message "An Unexpected Error Occurred. Error Details: $_.Exception.Message"
    }
}
}

```

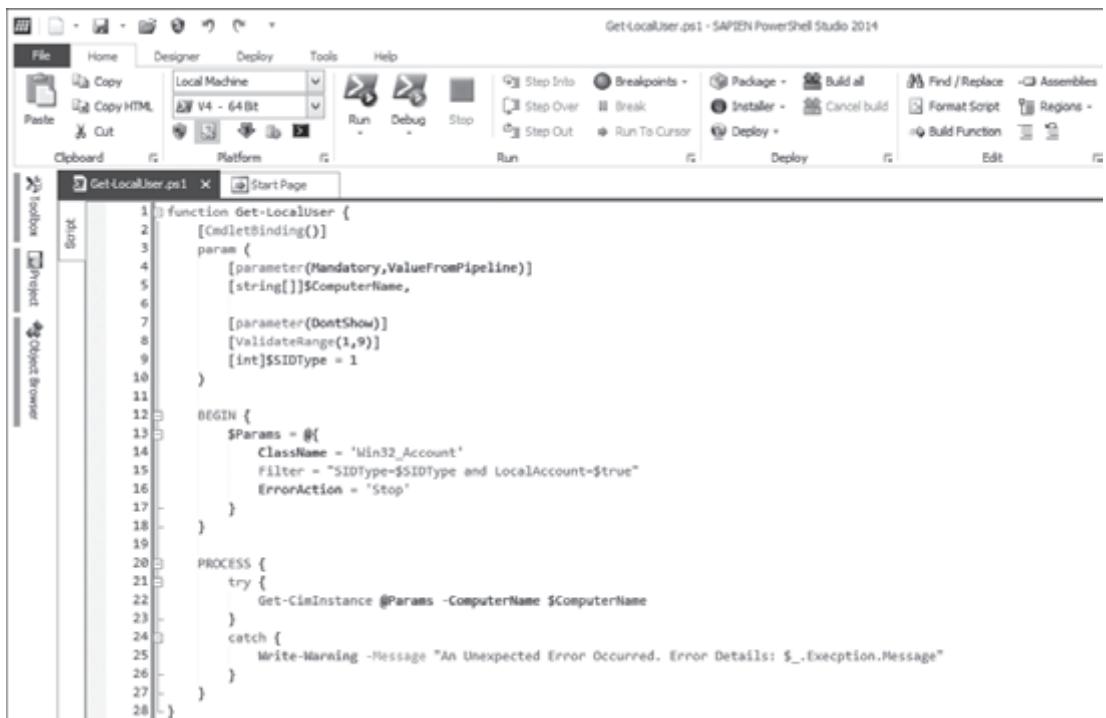


Figure 20-26

Notice how our code is cleanly formatted. We've also used something called Splatting, since all of the parameters that we wanted to specify on the line with the **Get-CimInstance** cmdlet were making it really long. We never ever use the backtick or grave accent character to extend a line to the next line, as that character alone is an error waiting to happen.

We won't go into Splatting in detail, but we recommend reading the `about_Splatting` Help topic in PowerShell, for more information.

Our function appears to work properly when we specify input via a parameter or the pipeline and we can see that it executes the code in the **Catch** block when an invalid computer name is specified.

```
. .\Get-LocalUser.ps1
Get-LocalUser -ComputerName sql01, web01

Get-LocalUser -ComputerName DoesNotExist
'DoesNotExist' | Get-LocalUser
```

The screenshot shows an Administrator Windows PowerShell window with the following content:

```
Administrator: Windows PowerShell
PS C:\Scripts> . .\Get-LocalUser.ps1
PS C:\Scripts> Get-LocalUser -ComputerName sql01, web01
Name          Caption        AccountType      SID           Domain       PSComputerName
---          ---           ---           ---           ---           ---
Administrator SQL01\Administrator 512   S-1-5-21-42855845... SQL01      sql01
Guest         SQL01\Guest     512   S-1-5-21-42855845... SQL01      sql01
Administrator WEB01\Administrator 512   S-1-5-21-21943053... WEB01     web01
Guest         WEB01\Guest     512   S-1-5-21-21943053... WEB01     web01

PS C:\Scripts> 'sql01', 'web01' | Get-LocalUser
Name          Caption        AccountType      SID           Domain       PSComputerName
---          ---           ---           ---           ---           ---
Administrator SQL01\Administrator 512   S-1-5-21-42855845... SQL01      sql01
Guest         SQL01\Guest     512   S-1-5-21-42855845... SQL01      sql01
Administrator WEB01\Administrator 512   S-1-5-21-21943053... WEB01     web01
Guest         WEB01\Guest     512   S-1-5-21-21943053... WEB01     web01

PS C:\Scripts> Get-LocalUser -ComputerName DoesNotExist
WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message
PS C:\Scripts> 'DoesNotExist' | Get-LocalUser
WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message
PS C:\Scripts> _
```

Figure 20-27

There's a logic error with our `Get-LocalUser` function, though. Can you identify the problem? Logic errors can be much more difficult to find than syntax errors, because as with our function, they sometimes go undetected until they're used in a specific scenario where the logic problem is finally exposed.

The problem with our `Get-LocalUser` function is that we're using `Get-CimInstance` to run in parallel against multiple computers. That by itself isn't an issue, but once we turn our non-terminating error into a terminating error when one of the computers can't be reached, it makes the entire command fail. That's definitely not what we had in mind when we designed this function. Execute it against a hundred computers and have just one fail, and that one fail will cause the entire list of computers to fail.

```
Get-LocalUser -ComputerName sql01, web01, DoesNotExist
'sql01', 'web01', 'DoesNotExist' | Get-LocalUser
```

```

Administrator: Windows PowerShell
PS C:\Scripts> Get-LocalUser -ComputerName sql01, web01, DoesNotExist
WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred
while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the
network and that the name provided is spelled correctly..Exception.Message
PS C:\Scripts> sql01, web01, DoesNotExist | Get-LocalUser
Name          Caption      AccountType   SID           Domain       PSComputerName
----          -----      -----        ---           -----       -----
Administrator  SQL01\Administrator  S12          S-1-5-21-42855845...  SQL01      sql01
Guest          SQL01\Guest        S12          S-1-5-21-42855845...  SQL01      sql01
Administrator  WEB01\Administrator  S12          S-1-5-21-21943053...  WEB01      web01
Guest          WEB01\Guest        S12          S-1-5-21-21943053...  WEB01      web01
WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred
while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the
network and that the name provided is spelled correctly..Exception.Message
PS C:\Scripts>

```

Figure 20-28

Notice in Figure 20-28 that the issue only occurs when the computers are specified via parameter input, but it works the way we designed it to when using pipeline input. To fully understand the problem, you'll need to understand how pipeline input versus parameter input works. When parameter input is used, the **Process** block runs a total of one time, regardless if one or 100 computers are specified. However, if they are sent in via pipeline input, the **Process** block runs one time for each item sent through the pipeline.

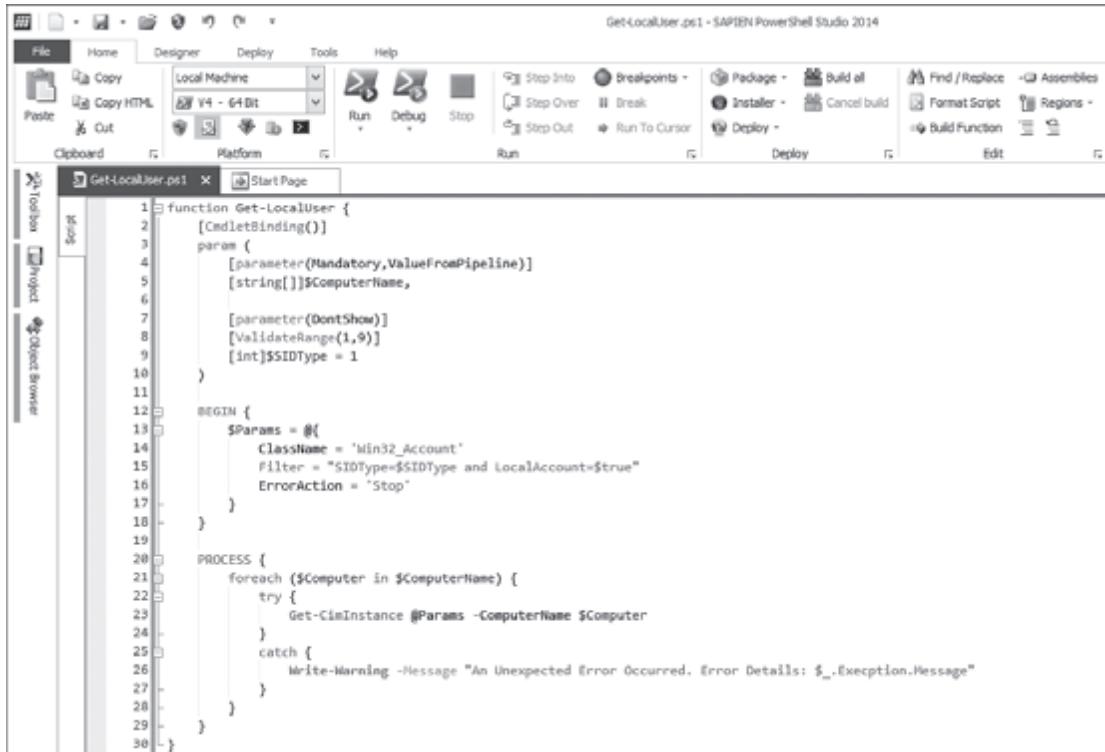
To correct this logic problem, we need the code in the **Process** block to run one time for each item that is specified via parameter input, just like it does for pipeline input. In order to make this happen, we'll add a **Foreach** loop (the **Foreach** construct, not to be confused with the **ForEach-Object** cmdlet) inside the process block to iterate through the items in the **ComputerName** array.

```

function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1,9)]
        [int]$SIDType = 1
    )
    BEGIN {
        $Params = @{
            ClassName = 'Win32_Account'
            Filter = "SIDType=$SIDType and LocalAccount=$true"
            ErrorAction = 'Stop'
        }
    }
    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Get-CimInstance @Params -ComputerName $Computer
            }
            catch {
                Write-Warning -Message "An Unexpected Error Occurred. Error Details: $_.Exception.
Message"
            }
        }
    }
}

```

## Debugging and error handling



The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The title bar reads "Get-LocalUser.ps1 - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Copy, Paste, Run, Debug, Stop, Step Into, Breakpoints, Package, Build all, Find / Replace, Assemblies, and other development tools. The main code editor window displays the following PowerShell script:

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1,9)]
        [int]$SIDType = 1
    )
    BEGIN {
        $Params = @{
            Classname = 'Win32_Account'
            Filter = "SIDType=$SIDType and LocalAccount=$true"
            ErrorAction = 'Stop'
        }
    }
    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Get-CimInstance @Params -ComputerName $Computer
            }
            catch {
                Write-Warning -Message "An Unexpected Error Occurred. Error Details: $_.Exception.Message"
            }
        }
    }
}
```

Figure 20-29

Now that's more like it. Our function produces the same results, regardless of whether the pipeline or parameter input is used and regardless of whether all of the computers are reachable or not.

```
..\Get-LocalUser.ps1
Get-LocalUser -ComputerName sql01, web01, DoesNotExist

'sql01', 'web01', 'DoesNotExist' | Get-LocalUser
```

```

Administrator: Windows PowerShell
PS C:\Scripts> .\Get-LocalUser.ps1
PS C:\Scripts> Get-LocalUser -ComputerName sql01, web01, DoesNotExist
Name          Caption       AccountType      SID           Domain      PSComputerName
---          -----
Administrator SQL01\Administrator 512   S-1-5-21-42855845... SQL01      sql01
Guest         SQL01\Guest     512   S-1-5-21-42855845... SQL01      sql01
Administrator WEB01\Administrator 512   S-1-5-21-21943053... WEB01      web01
Guest         WEB01\Guest     512   S-1-5-21-21943053... WEB01      web01
WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message

PS C:\Scripts> 'sql01', 'web01', 'DoesNotExist' | Get-LocalUser
Name          Caption       AccountType      SID           Domain      PSComputerName
---          -----
Administrator SQL01\Administrator 512   S-1-5-21-42855845... SQL01      sql01
Guest         SQL01\Guest     512   S-1-5-21-42855845... SQL01      sql01
Administrator WEB01\Administrator 512   S-1-5-21-21943053... WEB01      web01
Guest         WEB01\Guest     512   S-1-5-21-21943053... WEB01      web01
WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message

PS C:\Scripts> _

```

Figure 20-30

## Displaying errors with Write-Warning and Write-Error

You probably noticed in the previous section of this chapter that we used the **Write-Warning** cmdlet to display warning messages when a computer is unreachable.

Similar to the **\$ErrorActionPreference** preference variable, there is also a **\$WarningPreference** variable that has a default value of **Continue**. The effect of the **Write-Warning** cmdlet can vary depending on the value assigned to this preference variable, and as with **\$ErrorActionPreference**, we recommend keeping the default setting. There are also **-WarningAction** and **-WarningVariable** parameters that can be used on a per command basis, if needed.

Refer to the previous section in this chapter to see demonstrations of the **Write-Warning** cmdlet. In addition to the **Write-Warning** cmdlet, there is a **Write-Error** cmdlet.

The **Write-Error** cmdlet is used to declare a non-terminating error. To demonstrate the **Write-Error** cmdlet, we'll simply replace **Write-Warning** with **Write-Error** in our function.

```

function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1, 9)]
        [int]$SIDType = 1
    )

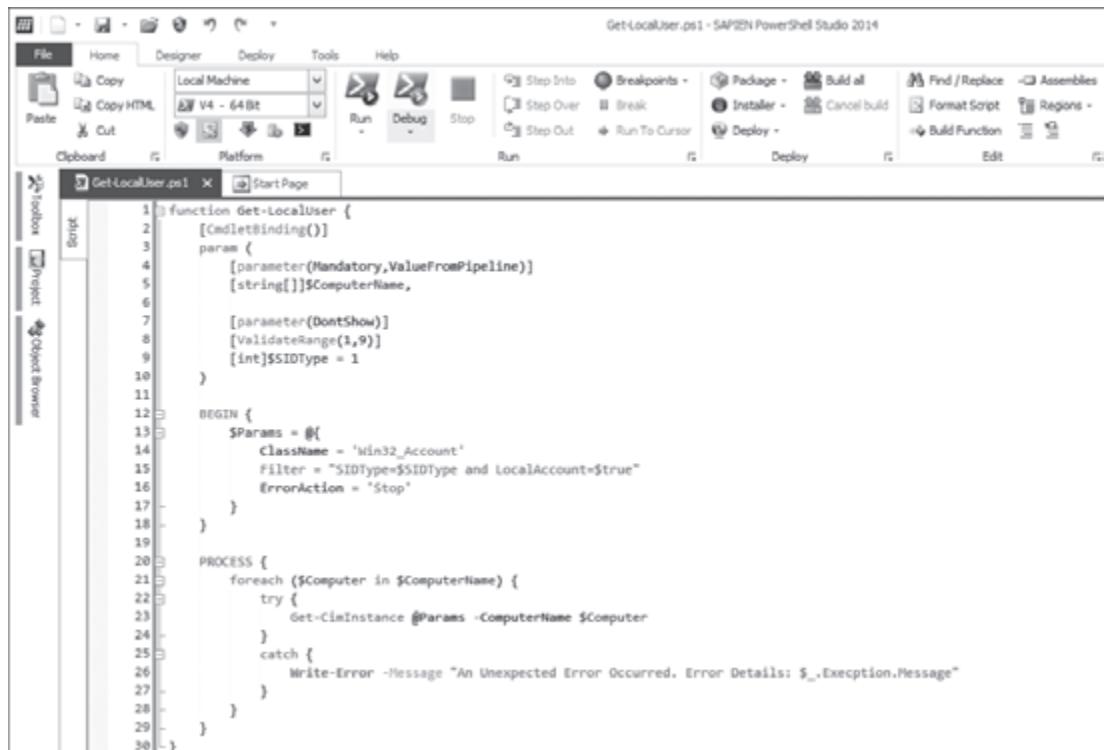
    BEGIN {
        $Params = @{
            ClassName = 'Win32_Account'
            Filter = "SIDType=$SIDType and LocalAccount=$true"
            ErrorAction = 'Stop'
        }
    }

    PROCESS {

```

## Debugging and error handling

```
foreach ($Computer in $ComputerName) {
    try {
        Get-CimInstance @Params -ComputerName $Computer
    }
    catch {
        Write-Error -Message "An Unexpected Error Occurred. Error Details: $_.Exception.Message"
    }
}
```



```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1,9)]
        [int]$SIDType = 1
    )
    BEGIN {
        $Params = @{
            ClassName = 'Win32_Account'
            Filter = "SIDType=$SIDType and LocalAccount=$true"
            ErrorAction = 'Stop'
        }
    }
    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Get-CimInstance @Params -ComputerName $Computer
            }
            catch {
                Write-Error -Message "An Unexpected Error Occurred. Error Details: $_.Exception.Message"
            }
        }
    }
}
```

Figure 20-31

```
Get-LocalUser -ComputerName 'sql01', 'web01', 'DoesNotExist'
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "Get-LocalUser -ComputerName 'sql01', 'web01', 'DoesNotExist'". The output table has columns: Name, Caption, AccountType, SID, Domain, and PSComputerName. It lists four entries: Administrator (SID 512), Guest (SID 512), and two more entries for 'sql01' and 'web01'. An error message is displayed at the bottom of the output:

```
Get-LocalUser : An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message
At line:1 char:1
+ Get-LocalUser -ComputerName 'sql01', 'web01', 'DoesNotExist'
+ ~~~~~~
+ CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Get-LocalUser
```

Figure 20-32

## Storing errors in a log for later

We plan to schedule our **Get-LocalUser** function to run against hundreds of computers. We need to log the computer names that it fails on so that we can retry those later, in case they were turned off when the function tried to query them. We also want to log the details of the errors, but to a different log file, just in case we want to review them later.

We've added a couple of new variables. One is named **ErrorSummaryLog** and the other is named **ErrorDetailLog**. We remove the previous log files, if they exist, in our **Begin** block, and then we write each failed computer's name to the log file in our **Catch** block, which is the easy part. There's something that we want you to be aware of: alternate data streams are data streams other than the default success output data stream to include errors, warnings, verbose, and debug output. Warnings are written to an alternate data stream and **Write-Warning** displays them on screen for us, but since we want to use the **Tee-Object** cmdlet to write our output of **Write-Warning** to the console and to a file, we must use the **3>&1** redirection operator to send warnings and success output to the success output data stream.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1, 9)]
        [int]$SIDType = 1,

        [ValidateNotNullOrEmpty()]
        [string]$ErrorSummaryLog = 'C:\Scripts\ErrorSummaryLog.txt',

        [ValidateNotNullOrEmpty()]
        [string]$ErrorDetailLog = 'C:\Scripts\ErrorDetailLog.txt'
    )

    BEGIN {
        $Params = @{
            ClassName = 'Win32_Account'
            Filter = "SIDType=$SIDType and LocalAccount=$true"
            ErrorAction = 'Stop'
            ErrorVariable = 'ErrorDetail'
        }
    }
```

### Debugging and error handling

```
Remove-Item -Path $ErrorSummaryLog, $EnableErrorDetailLog -ErrorAction SilentlyContinue
}

PROCESS {
    foreach ($Computer in $ComputerName) {
        try {
            Get-CimInstance @Params -ComputerName $Computer
        }
        catch {
            $Computer | Out-File -FilePath $ErrorSummaryLog -Append

            Write-Warning -Message "An Unexpected Error Occured. Error Details: $_.Exception.
Message" 3>&1 | Tee-Object -FilePath $ErrorDetailLog -Append
        }
    }
}
```

The screenshot shows the SAPiEN PowerShell Studio 2014 interface with the script 'Get-LocalUser.ps1' open. The code editor displays the PowerShell script with syntax highlighting. The script defines a function 'Get-LocalUser' with parameters for computer names, SID type, and log files. It uses a 'try-catch' block to handle errors, writing them to a summary log and a detailed log. The SAPiEN interface includes toolbars, a ribbon menu, and various project-related tabs.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1,9)]
        [int]$SIDType = 1,
        [ValidateNotNullOrEmpty()]
        [string]$ErrorSummaryLog = 'C:\Scripts\ErrorSummaryLog.txt',
        [ValidateNotNullOrEmpty()]
        [string]$ErrorDetailLog = 'C:\Scripts\ErrorDetailLog.txt'
    )

    BEGIN {
        $Params = @{
            ClassName = 'Win32_Account'
            Filter = "SIDType=$SIDType and LocalAccount=$true"
            ErrorAction = 'Stop'
            ErrorVariable = 'ErrorDetail'
        }
    }

    Remove-Item -Path $ErrorSummaryLog, $EnableErrorDetailLog -ErrorAction SilentlyContinue
}

PROCESS {
    foreach ($Computer in $ComputerName) {
        try {
            Get-CimInstance @Params -ComputerName $Computer
        }
        catch {
            $Computer | Out-File -FilePath $ErrorSummaryLog -Append

            Write-Warning -Message "An Unexpected Error Occured. Error Details: $_.Exception.Message" 3>&1 |
Tee-Object -FilePath $ErrorDetailLog -Append
        }
    }
}
```

Figure 20-33

For more information about the different data streams and redirection operators, see the `about_Redirection` Help topic in PowerShell.

Once again, we dot source our script that contains our `Get-LocalUser` function, and then we run it against four servers, two of which do not exist.

```
Get-LocalUser -ComputerName sql01, DoesNotExist, web01, PoweredOff
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command run is "Get-LocalUser -ComputerName sql01, DoesNotExist, web01, PoweredOff". The output table includes columns: Name, Caption, AccountType, SID, Domain, and PSComputerName. The results for 'sql01' and 'web01' are as follows:

Name	Caption	AccountType	SID	Domain	PSComputerName
Administrator	SQL01\Administrator	512	S-1-5-21-42855845...	SQL01	sql01
Guest	SQL01\Guest	512	S-1-5-21-42855845...	SQL01	sql01

Below the table, several warning messages are displayed:

- WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message
- WARNING: Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message
- WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer PoweredOff. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message
- WARNING: Kerberos authentication: Cannot find the computer PoweredOff. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message

Figure 20-34

We'll receive the content of our file that contains the computers that failed.

```
Get-Content -Path .\ErrorSummaryLog.txt
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command run is "Get-Content -Path .\ErrorSummaryLog.txt". The output displays the contents of the file, which include the names of the failed computers: "DoesNotExist" and "PoweredOff".

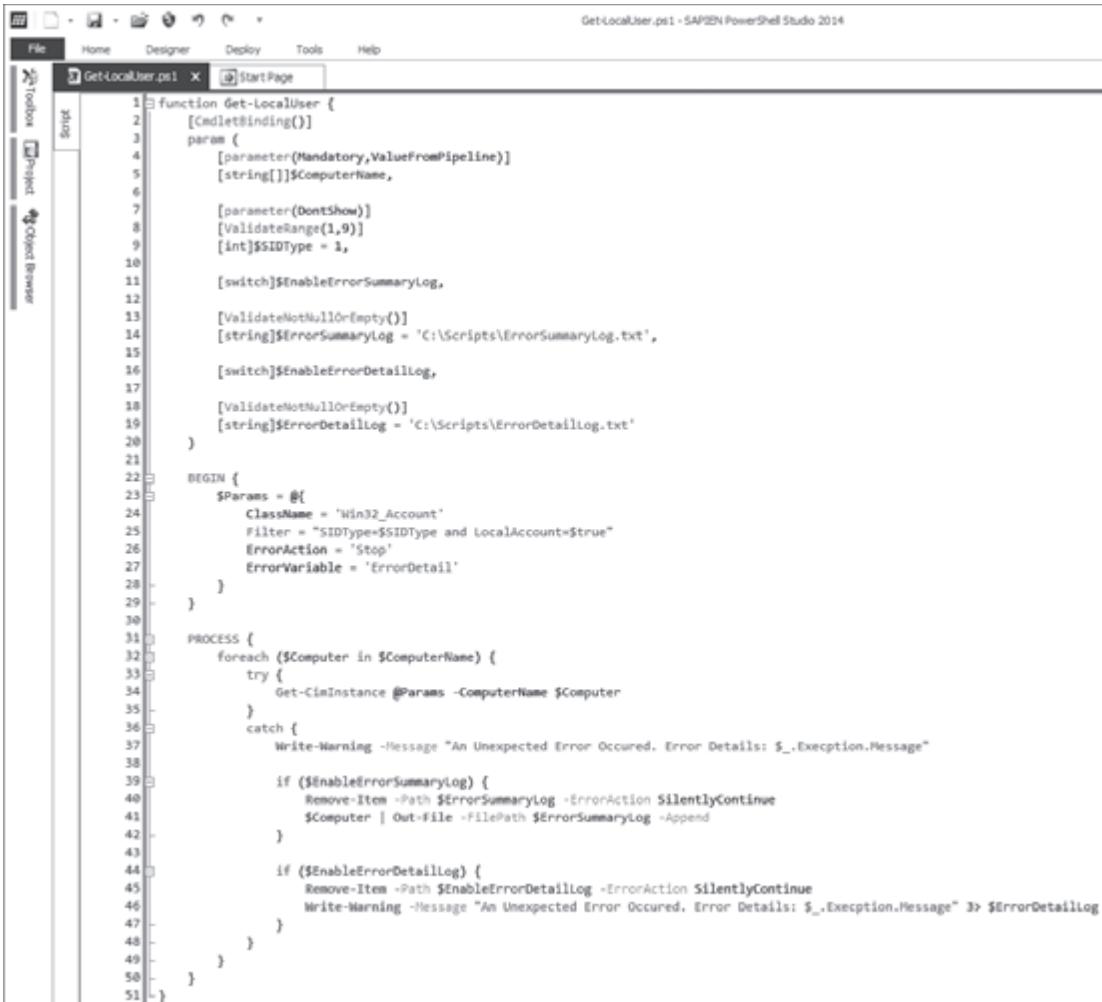
Figure 20-35

We want our tool to be a little more versatile. We don't always want to have to log the errors to a file and we want to be able to have a summary log or detailed log or both. However, we want to have choices without the person we plan to turn this tool over needing to modify the actual code to make this happen.

In order to add these capabilities, we've added two `Switch` parameters, `$EnableErrorSummaryLog` and `$EnableErrorDetailedLog`, which enable their corresponding error logs. We've also moved the code that deletes the previous log (if one exists) to the `If` blocks where we're controlling whether or not

the logs are enabled. That way, if we run only one of the logs, we're not deleting the previous log file from the other one, because the person running this tool may need that other log file and they probably wouldn't be too happy with us for deleting it.

```
function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1, 9)]
        [int]$SIDType = 1,
        [switch]$EnableErrorSummaryLog,
        [ValidateNotNullOrEmpty()]
        [string]$ErrorSummaryLog = 'C:\Scripts\ErrorSummaryLog.txt',
        [switch]$EnableErrorDetailLog,
        [ValidateNotNullOrEmpty()]
        [string]$ErrorDetailLog = 'C:\Scripts\ErrorDetailLog.txt'
    )
    BEGIN {
        $Params = @{
            ClassName = 'Win32_Account'
            Filter = "SIDType=$SIDType and LocalAccount=$true"
            ErrorAction = 'Stop'
            ErrorVariable = 'ErrorDetail'
        }
    }
    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Get-CimInstance @Params -ComputerName $Computer
            }
            catch {
                Write-Warning -Message "An Unexpected Error Occured. Error Details: $_.Exception.
Message"
                if ($EnableErrorSummaryLog) {
                    Remove-Item -Path $ErrorSummaryLog -ErrorAction SilentlyContinue
                    $Computer | Out-File -FilePath $ErrorSummaryLog -Append
                }
                if ($EnableErrorDetailLog) {
                    Remove-Item -Path $EnableErrorDetailLog -ErrorAction SilentlyContinue
                    Write-Warning -Message "An Unexpected Error Occured. Error Details: $_.Exception.
Message" 3> $ErrorDetailLog
                }
            }
        }
    }
}
```



```

function Get-LocalUser {
    [CmdletBinding()]
    param (
        [parameter(Mandatory,ValueFromPipeline)]
        [string[]]$ComputerName,
        [parameter(DontShow)]
        [ValidateRange(1,9)]
        [int]$SIDType = 1,
        [switch]$EnableErrorSummaryLog,
        [ValidateNotNullOrEmpty()]
        [string]$ErrorSummaryLog = 'C:\Scripts\ErrorSummaryLog.txt',
        [switch]$EnableErrorDetailLog,
        [ValidateNotNullOrEmpty()]
        [string]$ErrorDetailLog = 'C:\Scripts\ErrorDetailLog.txt'
    )

    BEGIN {
        $Params = @{
            ClassName = 'Win32_Account'
            Filter = "$SIDType=$SIDType and LocalAccount=$true"
            ErrorAction = 'Stop'
            ErrorVariable = 'ErrorDetail'
        }
    }

    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Get-CimInstance @Params -ComputerName $Computer
            }
            catch {
                Write-Warning -Message "An Unexpected Error Occurred. Error Details: $_.Exception.Message"

                if ($EnableErrorSummaryLog) {
                    Remove-Item -Path $ErrorSummaryLog -ErrorAction SilentlyContinue
                    $Computer | Out-File -FilePath $ErrorSummaryLog -Append
                }

                if ($EnableErrorDetailLog) {
                    Remove-Item -Path $ErrorDetailLog -ErrorAction SilentlyContinue
                    Write-Warning -Message "An Unexpected Error Occurred. Error Details: $_.Exception.Message" 3> $ErrorDetailLog
                }
            }
        }
    }
}

```

Figure 20-36

You can now use this tool by only specifying the **-ComputerName** parameter and no log will be generated. We can also enable either of the logs, individually or both, if needed. Also, notice the difference in the warning stream redirection operator **3>** to log the warnings to just a file.

```

Get-LocalUser -ComputerName sql01, DoesNotExist, web01, PoweredOff
Get-LocalUser -ComputerName sql01, DoesNotExist, web01, PoweredOff -EnableErrorSummaryLog
Get-LocalUser -ComputerName sql01, DoesNotExist, web01, PoweredOff -EnableErrorDetailLog

```

## Debugging and error handling

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It displays three separate command executions:

```
PS C:\Scripts> Get-LocalUser -ComputerName sql01, web01, DoesNotExist, PoweredOff
```

Name	Caption	AccountType	SID	Domain	PSComputerName
Administrator	SQL01\Administrator	512	S-1-5-21-42855845...	SQL01	sql01
Guest	SQL01\Guest	512	S-1-5-21-42855845...	SQL01	sql01
Administrator	WEB01\Administrator	512	S-1-5-21-21943053...	WEB01	web01
Guest	WEB01\Guest	512	S-1-5-21-21943053...	WEB01	web01

WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message

```
PS C:\Scripts> Get-LocalUser -ComputerName sql01, web01, DoesNotExist, PoweredOff -EnableErrorSummaryLog
```

Name	Caption	AccountType	SID	Domain	PSComputerName
Administrator	SQL01\Administrator	512	S-1-5-21-42855845...	SQL01	sql01
Guest	SQL01\Guest	512	S-1-5-21-42855845...	SQL01	sql01
Administrator	WEB01\Administrator	512	S-1-5-21-21943053...	WEB01	web01
Guest	WEB01\Guest	512	S-1-5-21-21943053...	WEB01	web01

WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message

WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer PoweredOff. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message

```
PS C:\Scripts> Get-LocalUser -ComputerName sql01, web01, DoesNotExist, PoweredOff -EnableErrorDetailLog
```

Name	Caption	AccountType	SID	Domain	PSComputerName
Administrator	SQL01\Administrator	512	S-1-5-21-42855845...	SQL01	sql01
Guest	SQL01\Guest	512	S-1-5-21-42855845...	SQL01	sql01
Administrator	WEB01\Administrator	512	S-1-5-21-21943053...	WEB01	web01
Guest	WEB01\Guest	512	S-1-5-21-21943053...	WEB01	web01

WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer DoesNotExist. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message

WARNING: An Unexpected Error Occurred. Error Details: WinRM cannot process the request. The following error occurred while using Kerberos authentication: Cannot find the computer PoweredOff. Verify that the computer exists on the network and that the name provided is spelled correctly..Exception.Message

Figure 20-37

In this chapter, we've given you a brief overview of how to debug your commands, as well as how to handle errors. An entire book could be written on debugging and error handling, so this certainly isn't a comprehensive reference on those subjects. However, we want you to be aware of the options that are available to you in PowerShell. With the discoverability that's built into PowerShell, you don't have to memorize everything about every cmdlet, keyword, and constraint—you just have to be aware that something exists, and then you can use the PowerShell Help system to figure out the details.

## Exercise 20 – Debugging and error handling

Several of the junior-level admins who use your tool that you created in the exercise of chapter 19 have reported receiving unhandled exceptions when using the tool.

### Task 1

Add error handling to the function that we've been working on in the exercises of the last few chapters. Ensure that a terminating error is generated.

### Task 2

Add the necessary code to have the function output debug information when the **-Debug** parameter is specified.

## Chapter 21

# Creating WinForm tools with PowerShell Studio

## What are Windows Forms?

Windows Forms, or WinForms, are part of the .NET Framework that allows developers to create graphical user interface (GUI) applications. The idea is that all the common graphical user interface controls, such as buttons and check boxes, are contained within the .NET Framework itself, so that all you have to do is leverage the application programming interface (API), which tells the .NET Framework where to put the various items, what size to make them, as well as any other detail you want to define. The .NET Framework itself takes care of drawing and managing all of these items on the screen for you.

Since Windows PowerShell is built on the .NET Framework, it has access to everything that WinForms has to offer, meaning you can use PowerShell to construct a complex and sophisticated GUI. PowerShell and GUIs—we know that those two terms sound like an oxymoron when used in the same sentence, but keep an open mind and you might be happily surprised by the time you complete this chapter.

## Making WinForms easy with SAPIEN PowerShell Studio

While it's possible to create a WinForms application or tool by only using PowerShell, it's what we would consider to be a difficult, error-prone, and a time-consuming trial and error process. To make a long story short, we didn't learn PowerShell to make our life more difficult or even to make GUIs. We learned PowerShell to make our life easier and while we could just use the PowerShell console to

achieve the task at hand or write a reusable tool using PowerShell Studio for other IT professionals to consume, sometimes the requirements of a task specifies that there must be a GUI. So, we'll leverage our existing skill set and write a GUI on top of our PowerShell code.

We can take our existing knowledge of PowerShell, along with tools that we've already created, and write GUIs on top of them very easily with PowerShell Studio.

## Using SAPIEN PowerShell Studio—A practical example

The Remote Server Administration Tools (RSAT) are required for this scenario, specifically the Active Directory PowerShell module.

To create a WinForms application in PowerShell Studio, go to the **New** drop-down menu and select **New Form**.

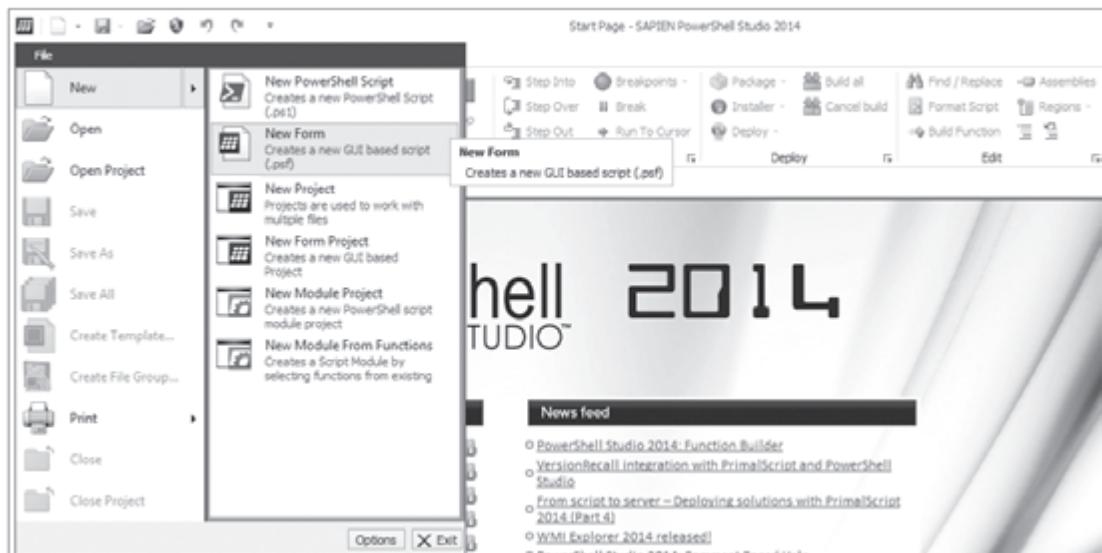


Figure 21-1

There are several predefined templates available for creating a WinForms application, but we'll begin with an **Empty Form** for the application that we'll be building in this chapter.

I guess we need to be careful of what terminology we use—we're building a tool (not an application). If we call it an application, people might think we're some kind of DevOps guru and want to give us a promotion and a raise.

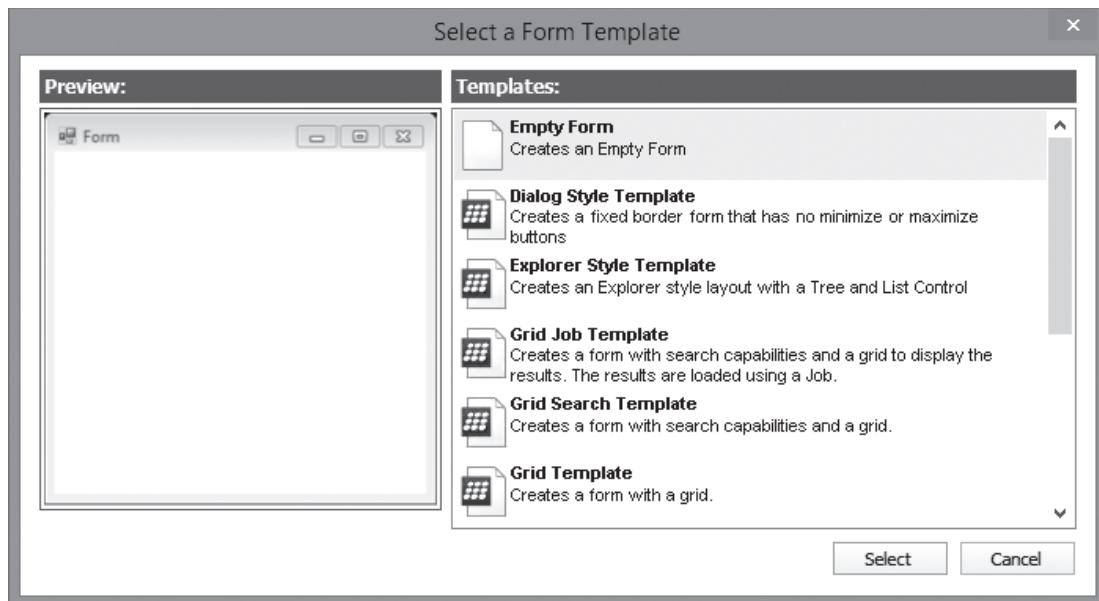


Figure 21-2

As we've said in previous chapters, you don't have to be a developer to use PowerShell, but if you used a version of Visual Studio in the past ten years, the interface in PowerShell Studio will look familiar. Don't get us wrong, PowerShell Studio isn't Visual Studio, but it includes items that are common among all Integrated Development Environments (IDEs). PowerShell Studio is technically an Integrated Scripting Environment (ISE), although we consider it to be a hybrid of the two, depending on what type of item you choose to create with it.

## Common controls

After selecting the **Empty Form** template in the previous step, we end up on the **Designer** tab within PowerShell Studio, where we can see our new form, along with a toolbox that contains controls that can be added to the form.

## Windows PowerShell: TFM

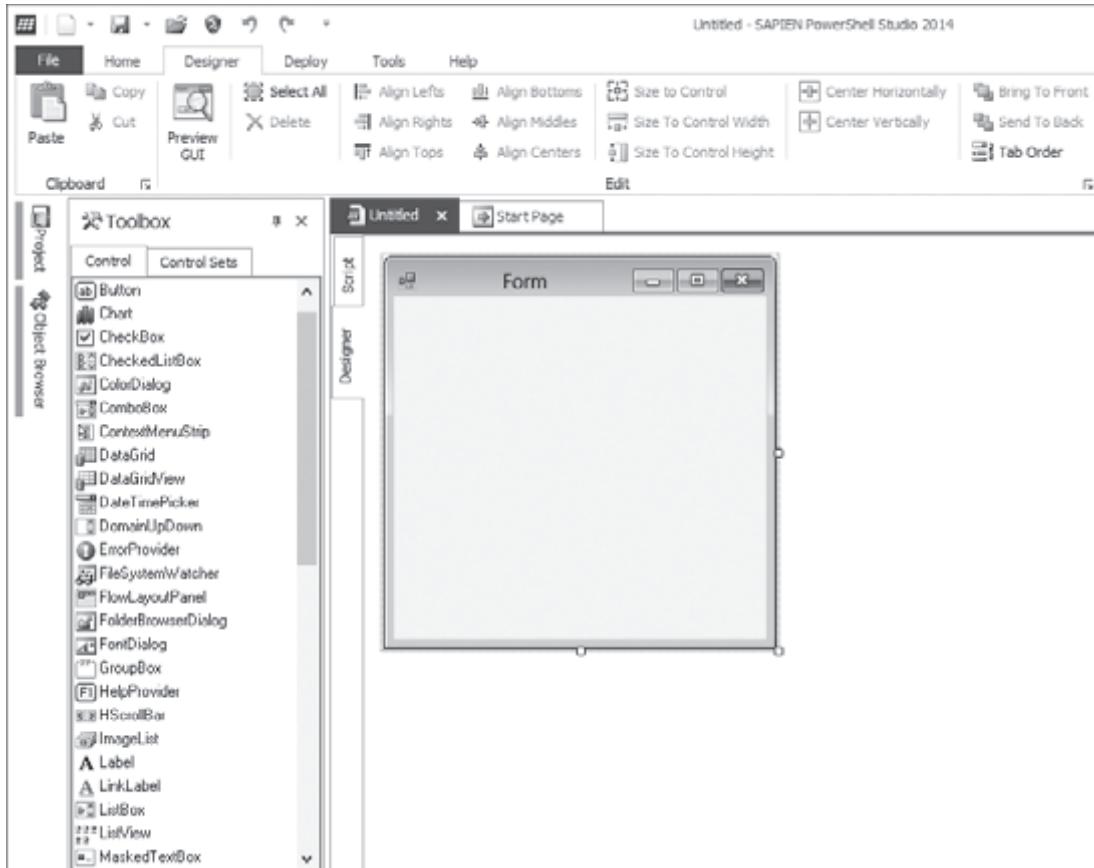


Figure 21-3

We drag a couple of labels, a couple of text boxes, and a button out of the toolbox and onto the form.

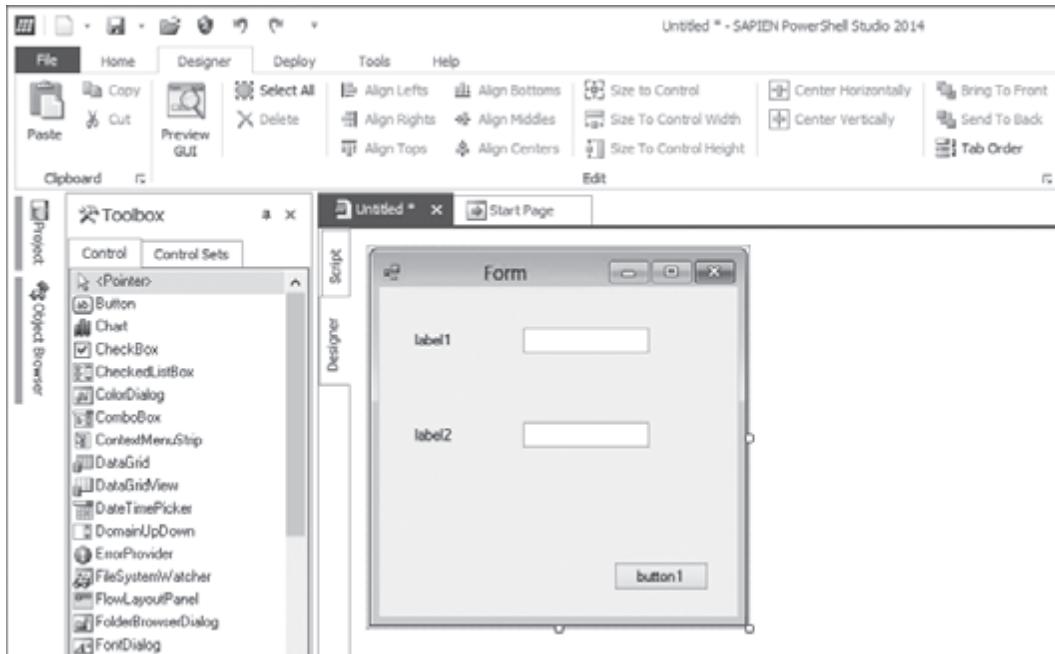


Figure 21-4

To change the **Text** property of a label, click on the label and type a name. In Figure 21-5, we change the **Text** property to “UserName”.

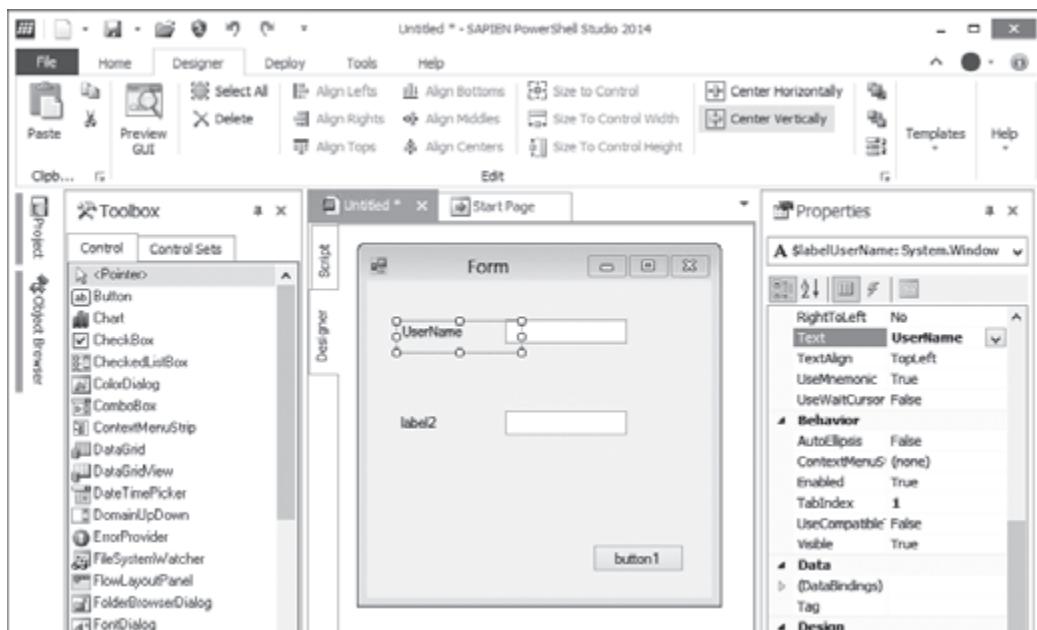


Figure 21-5

Notice that the name of the label, which is located in the **Properties** panel, automatically changes as well. The name of the label is now **labelUserName**.

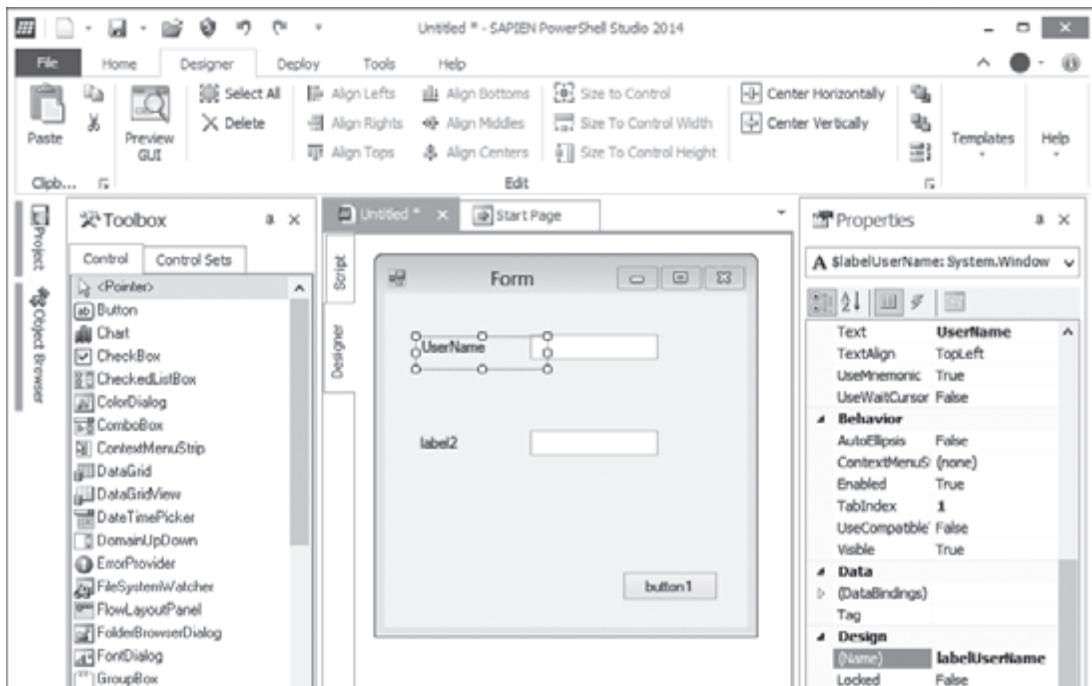


Figure 21-6

It's a best practice to change the name of any of the controls that have still their default name to something meaningful. An example of a control with its default name is the first textbox that is added to a form, by default, is called **textbox1**. Controls that you've changed the text property of, such as the label that we changed to "UserName" in Figure 21-5, will automatically be changed and already have a meaningful name.

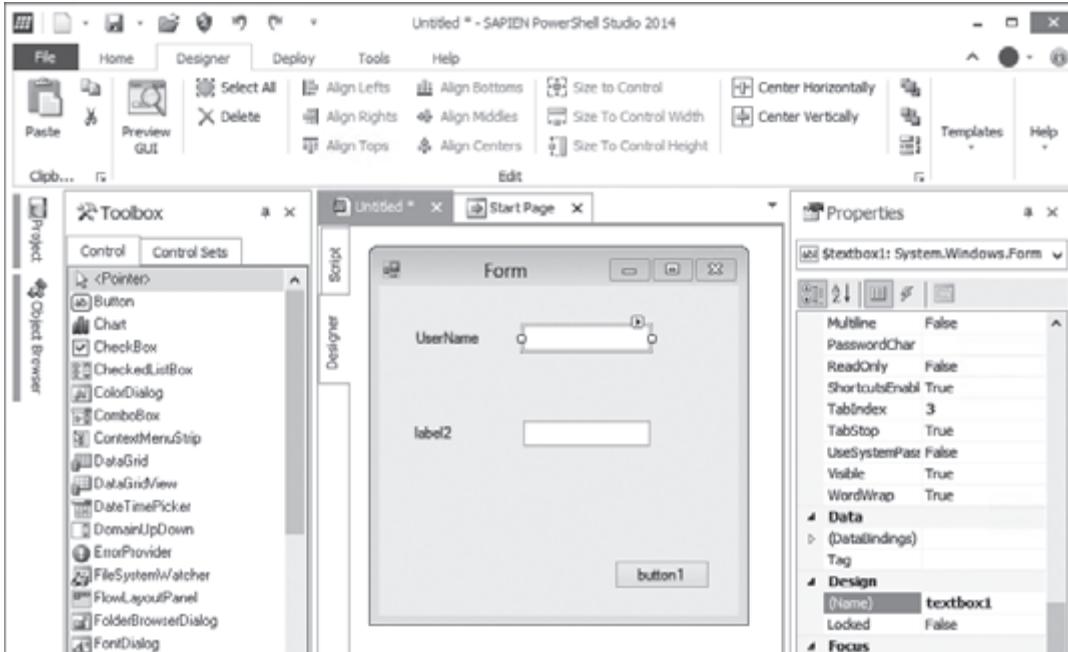


Figure 21-7

I'm sure you can image how difficult it would be to keep track of several different text boxes if the only difference in them is the number on the end of their name. We'll change the name of **textBox1** to **textBoxUserName**, so that it will be easier to identify when we're referencing it in our PowerShell code.

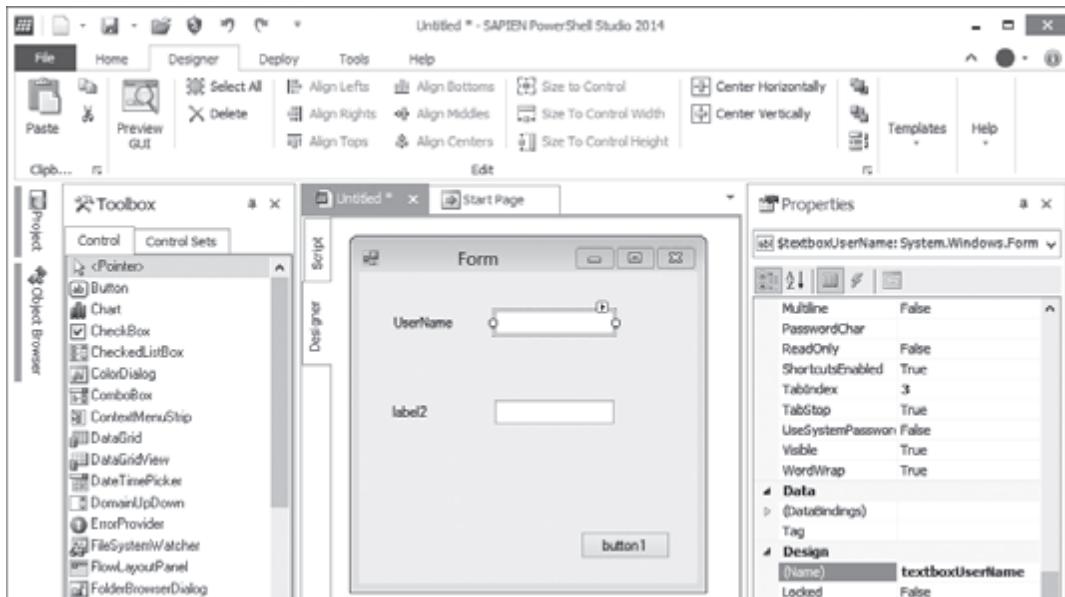


Figure 21-8

## Windows PowerShell: TFM

We went ahead and made the same changes for the second label and textbox, as shown in Figure 21-9.

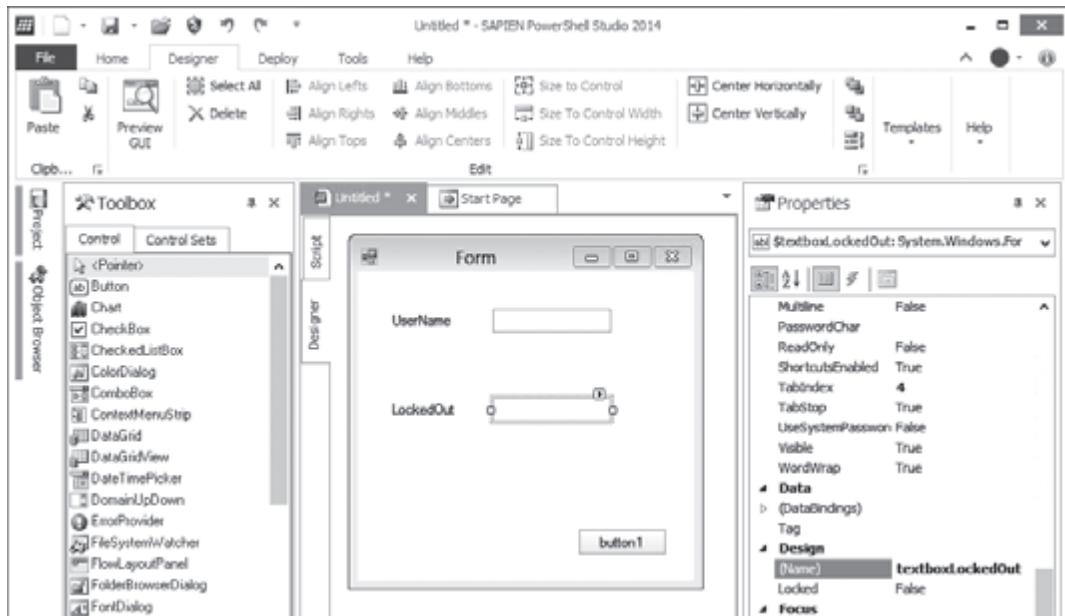


Figure 21-9

We need to change the text that's displayed on **button1** to something more meaningful as well, so we'll change it to the word "Go".

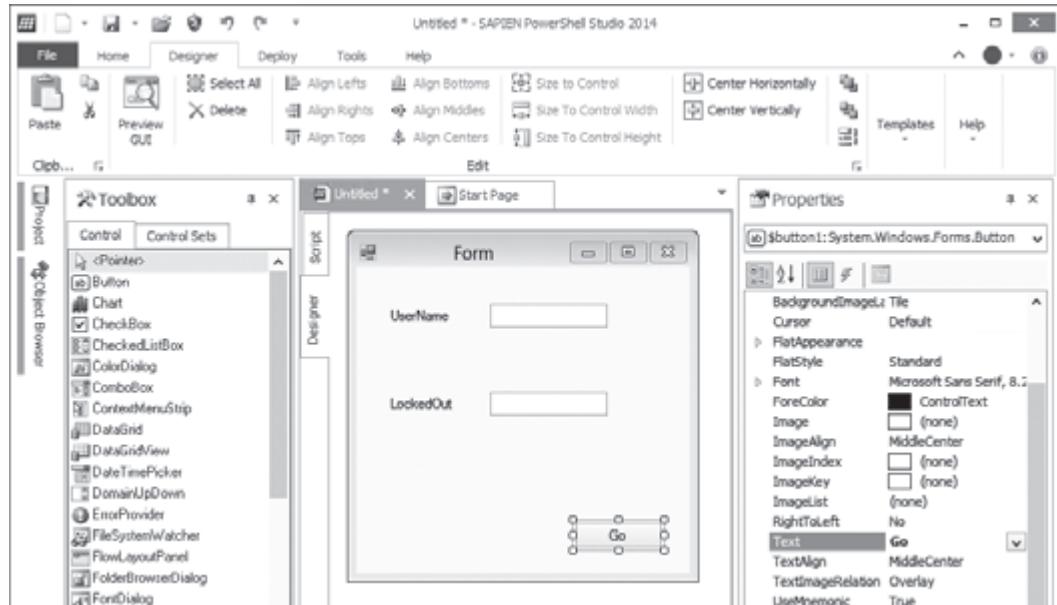


Figure 21-10

As with the labels, changing the text that's displayed on the button also changes its name. Since we changed the text of our button to "Go", the name of the button is now **buttonGo**.

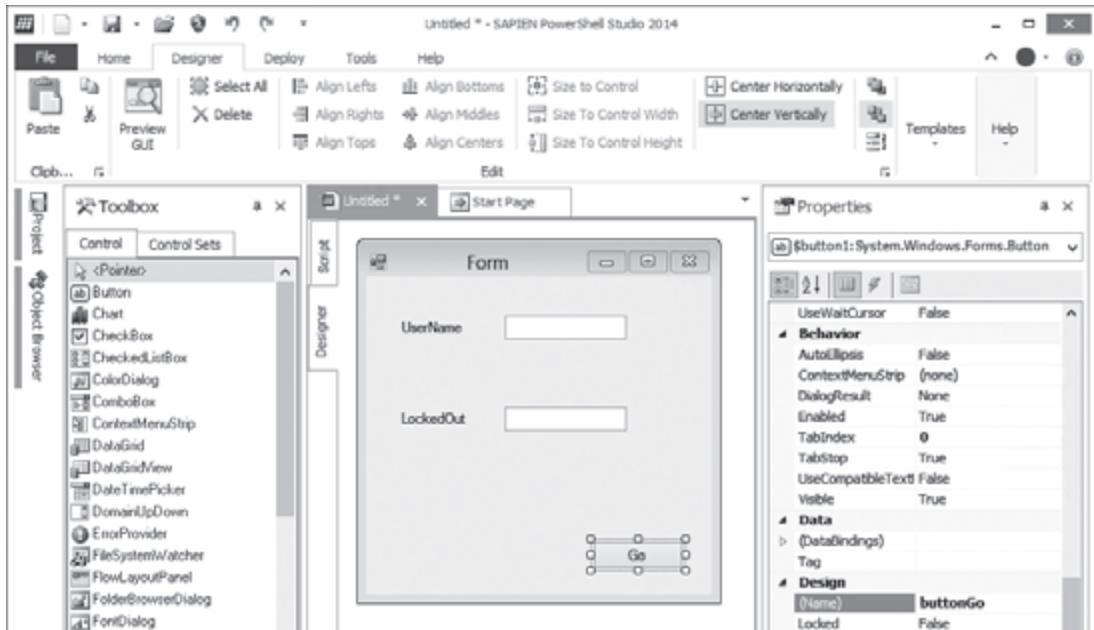


Figure 21-11

We will now select the form itself and change its text to something more meaningful for the user who will be using our tool. We'll use "Unlock User Account".

## Windows PowerShell: TFM

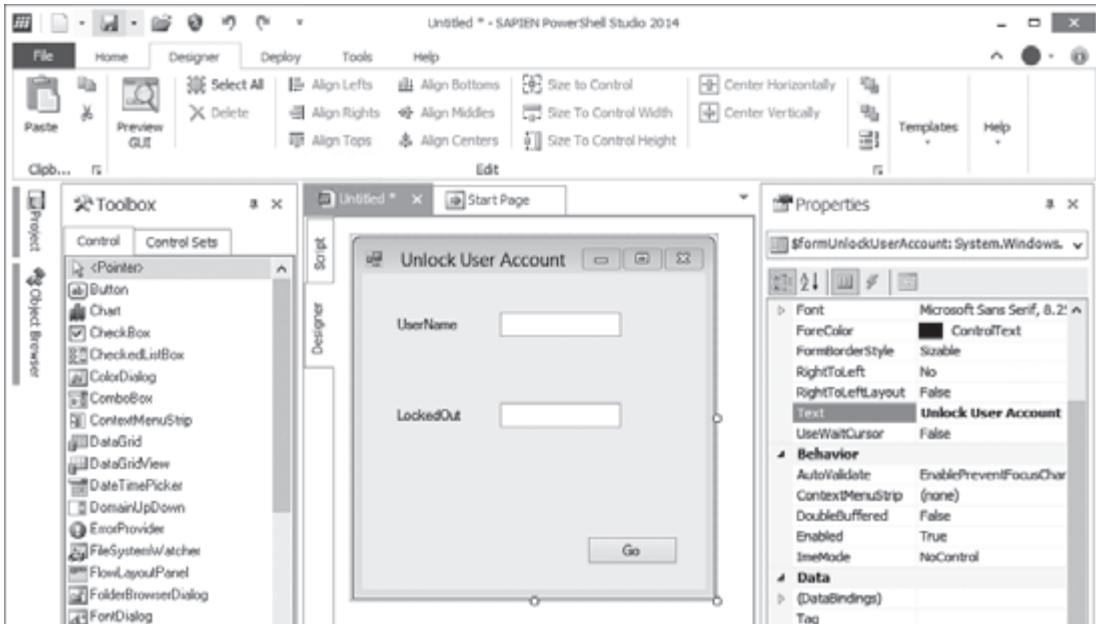


Figure 21-12

As with the previous controls, changing the value of the **Text** property also changes the name of the control. Notice that even though we've used spaces for the text value, the name value does not contain spaces.

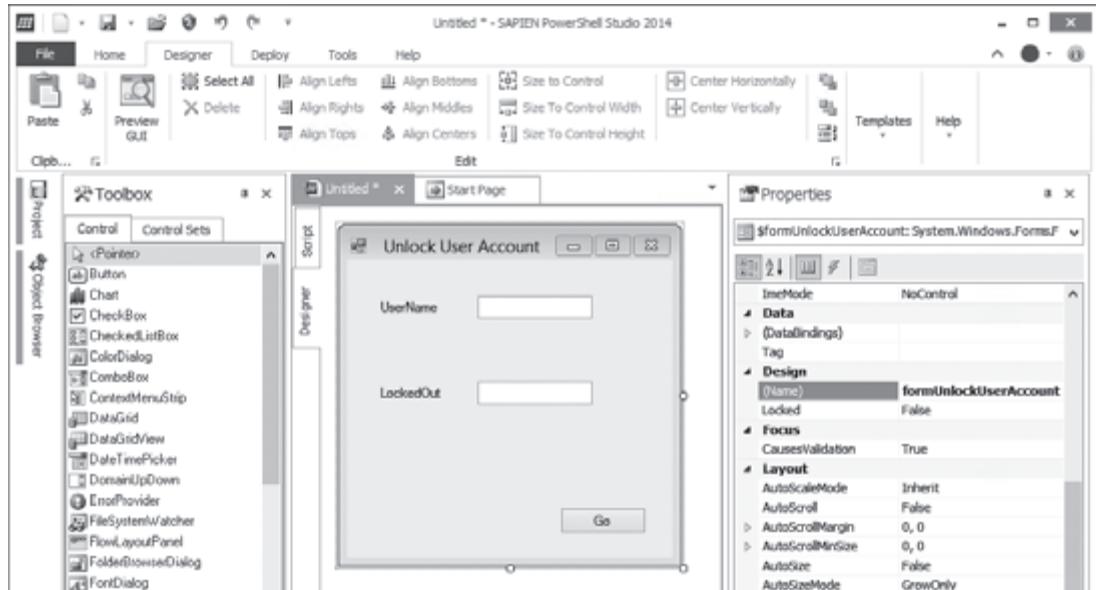


Figure 21-13

## Events and controls

Our form is now beginning to look like a real application, but it doesn't do anything at this point. If you ran it, entered some information into the textboxes and clicked **Go**, nothing would happen.

When designing WinForms, an event is an action that happens, for example, when you click the **Go** button. To wire up the default event handler of the **Go** button, double-click **Go**.

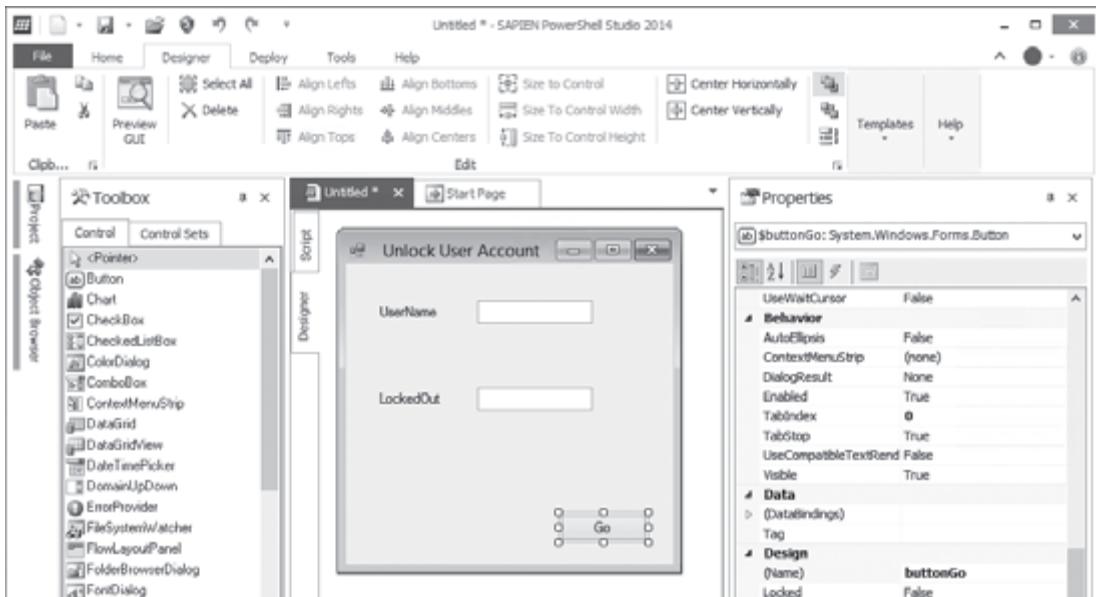


Figure 21-14

By double-clicking **Go**, we're taken to the **Script** tab of PowerShell Studio's interface. Depending on what resolution you're monitor is running, you may or may not have a lot of room on the screen to be able to work on the PowerShell code.

## Windows PowerShell: TFM

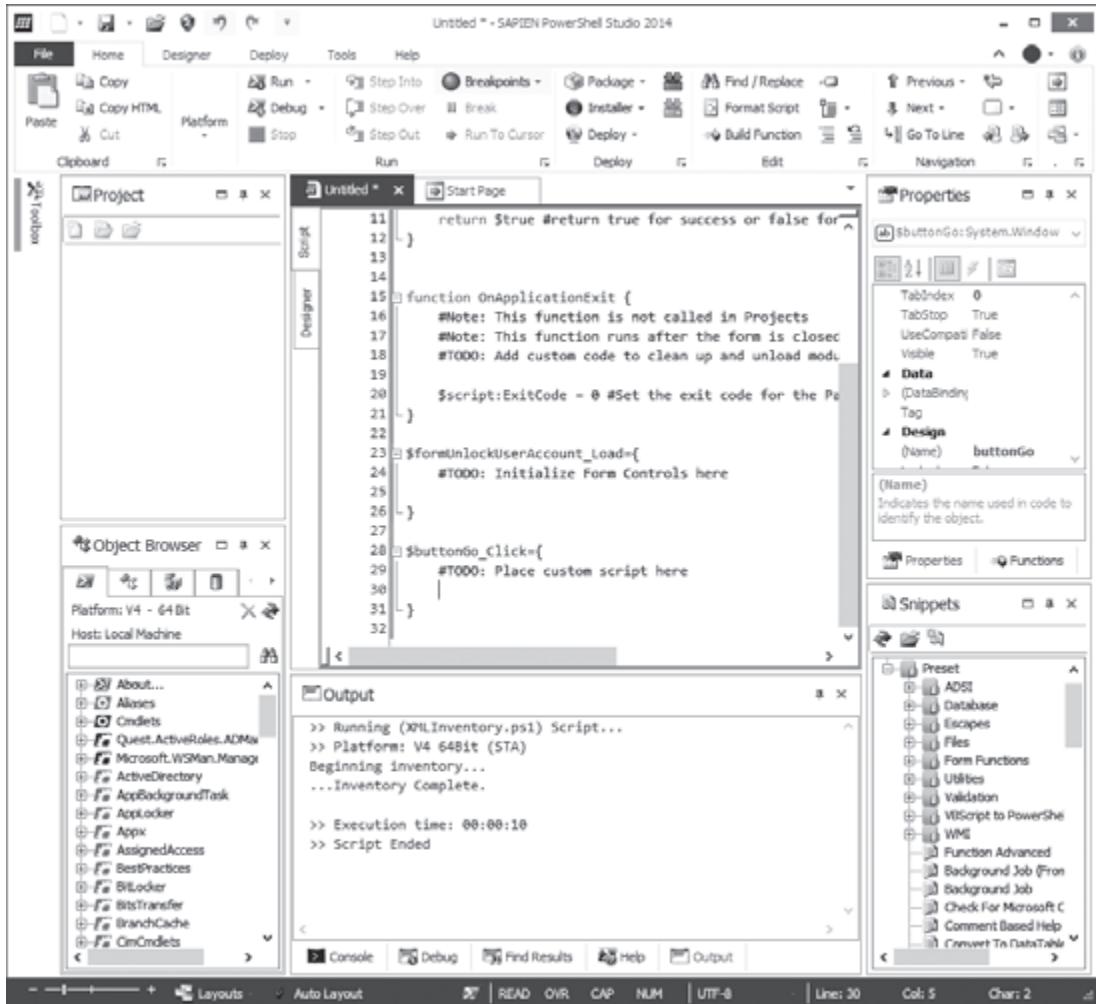


Figure 21-15

To alleviate this problem, change the interface to something with fewer panels displayed, by selecting the **Editor Only Layout** from the **Layout** icon on the bottom left side of PowerShell Studio, as shown in Figure 21-16.

## Creating WinForm tools with PowerShell Studio

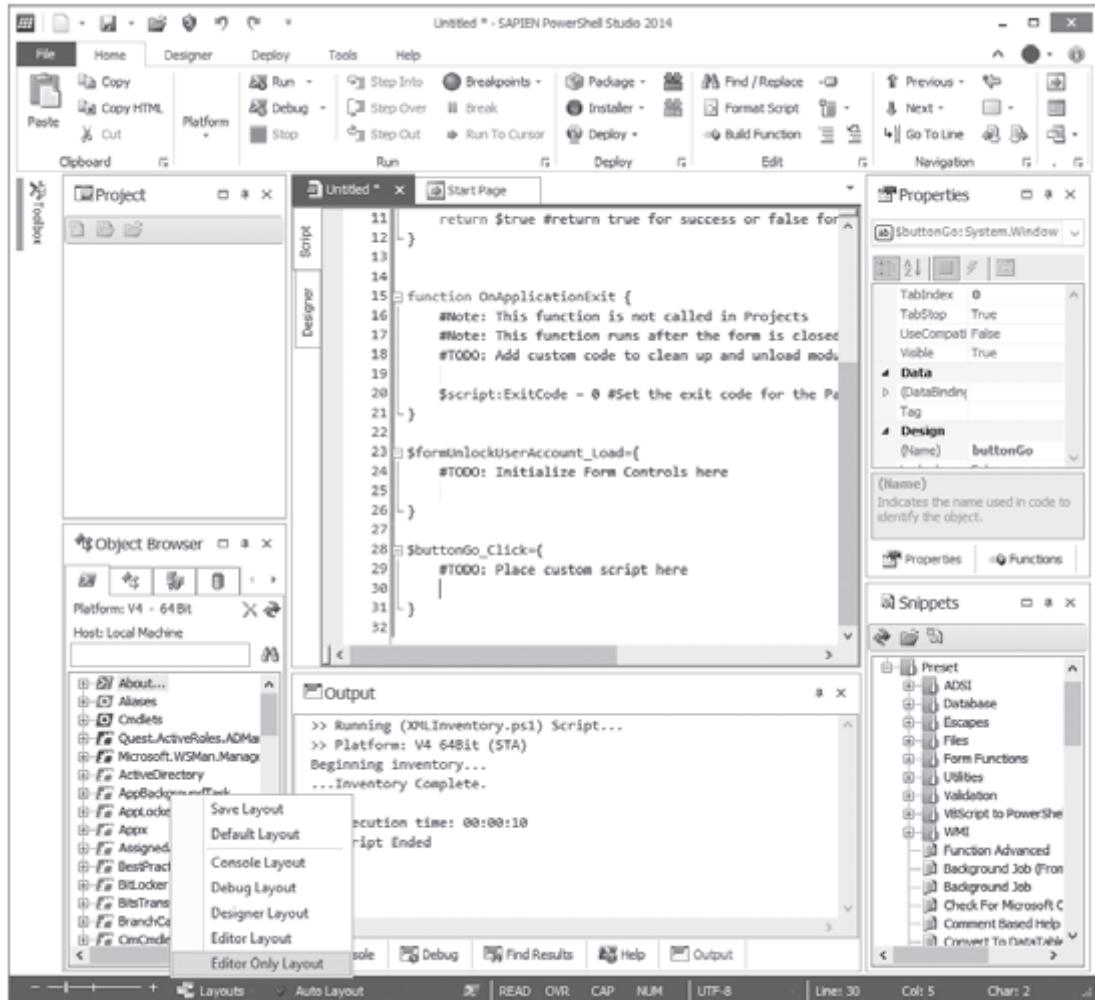


Figure 21-16

This will auto-hide all of the controls and give us more room to work on our PowerShell code, since we don't need those controls right now, anyway.

The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The title bar reads "Untitled - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Copy, Paste, Cut, and various deployment options like Run, Debug, Breakpoints, Package, Installer, Find/Replace, Format Script, Deploy, and Build Function. The main window has tabs for "Start Page" and "Script View". The "Script View" tab is active, displaying the following PowerShell script:

```

1
2
3 function OnApplicationLoad {
4     #Note: This function is not called in Projects
5     #Note: This function runs before the form is created
6     #Note: To get the script directory in the Packager user: Split-Path $hostInvocation.MyCommand.Path
7     #Note: To get the console output in the Packager (Windows Mode) use: $ConsoleOutput (Type: System.Collections.Generic.List`1[System.String])
8     #Important: Form controls cannot be accessed in this function
9     #TODO: Add modules and custom code to validate the application load
10
11     return $true #return true for success or false for failure
12 }
13
14
15 function OnApplicationExit {
16     #Note: This function is not called in Projects
17     #Note: This function runs after the form is closed
18     #TODO: Add custom code to clean up and unload modules when the application exits
19
20     $script:ExitCode = 0 #Set the exit code for the Packager
21 }
22
23 $formUnlockUserAccount_Load={
24     #TODO: Initialize Form Controls here
25
26 }
27
28 $buttonGo_Click={
29     #TODO: Place custom script here
30     |
31 }
32

```

The status bar at the bottom shows "Line: 30" and "Col: 5".

Figure 21-17

When we double-clicked **Go**, a `$buttonGo_Click` event handler was created, we were switched to the **Script View** tab in PowerShell Studio, and our cursor was automatically placed within the curly braces of this newly created event handler.

Whatever PowerShell code is placed inside of that particular event handler will be executed when **Go** is clicked on this form.

We've manually entered the line of code shown in Figure 21-18, which returns the `LockedOut` property for one of our users. We're hard coding everything in our line of code at this point, since we're prototyping and we want to make sure it's basic functionality works before adding any additional complexity that might make it more difficult to troubleshoot. For a basic test to verify our syntax is correct, we highlight our single line of code and press **Ctrl + Shift + F8** to run only that selection of code and have the results display in the PowerShell Studio console window, without having to save it first.

```
Get-ADUser -Identity 'jasonh' -Properties LockedOut | Select-Object -Property LockedOut
```

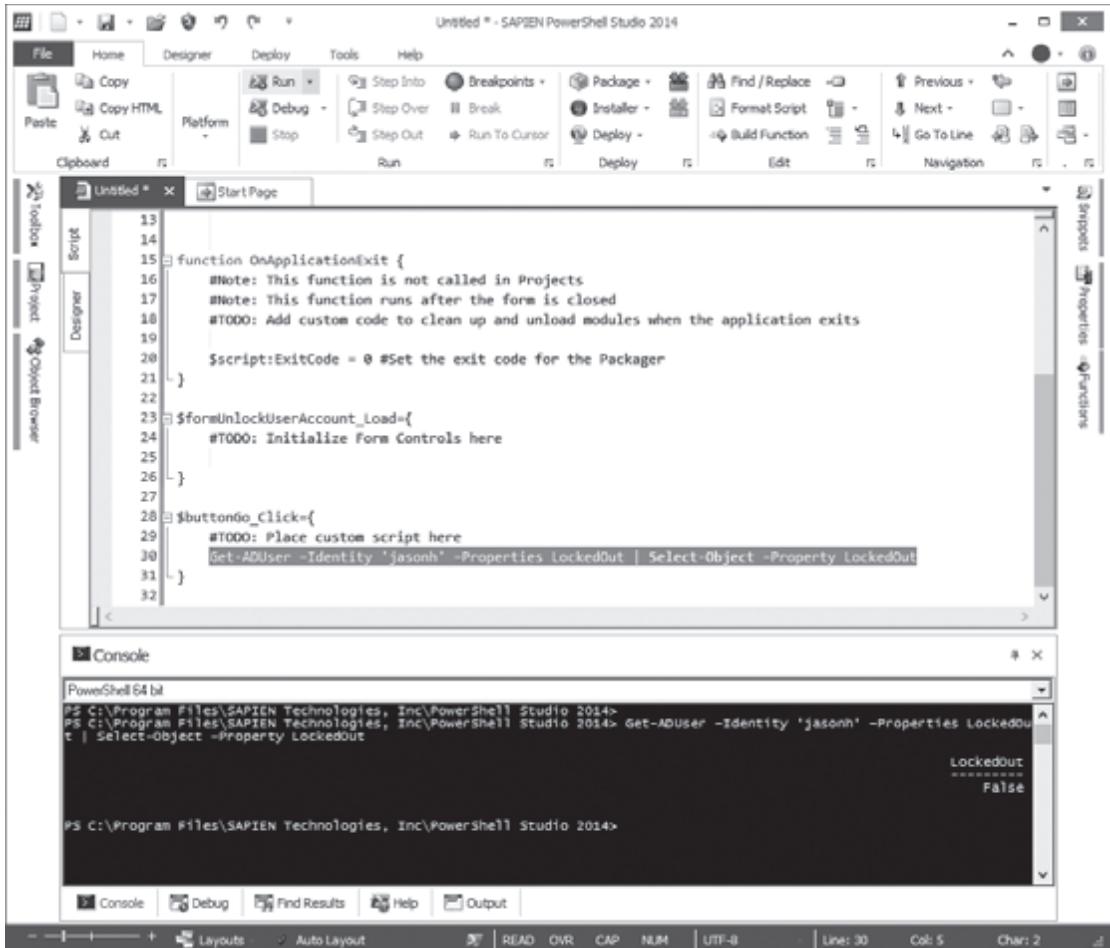


Figure 21-18

We could have also used **Ctrl + F8** to run our line of code without having to save it, if we wanted the results displayed in the **Output** pane instead.

I had a chance to ask one of the lead developers of PowerShell Studio why you're required to save your code before running it, unlike in the PowerShell ISE. One of the reasons was because of the following scenario: what if the script crashed the ISE, or in this case PowerShell Studio, and you lost all of your work? I hadn't thought of that and really never had any problems with the ISE crashing, but I have since experienced that issue in the ISE.

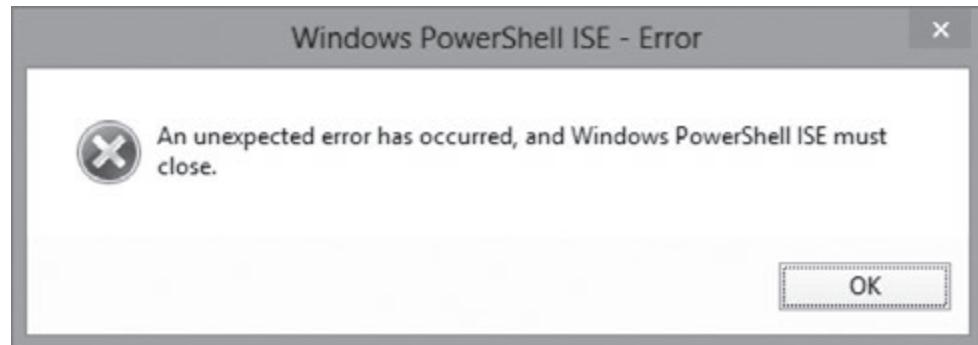


Figure 21-19

Now that we know that our line of code works, we'll start back at the beginning of that line of code and start typing the name of our **LockedOut** textbox, preceded by a \$. PrimalSense shows our available options and we select **textboxLockedOut**.

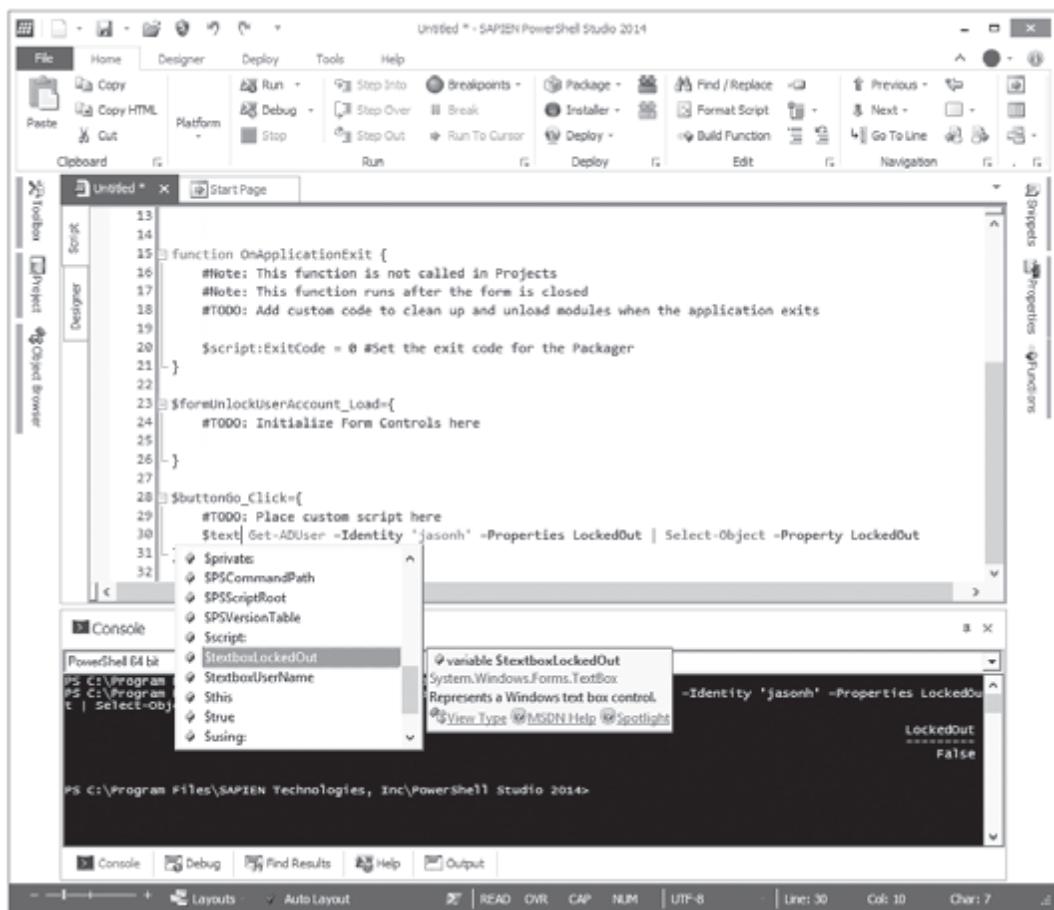


Figure 21-20

We type the dot character after the textbox name and start typing the word text. Once again, PrimalSense shows us our available options.

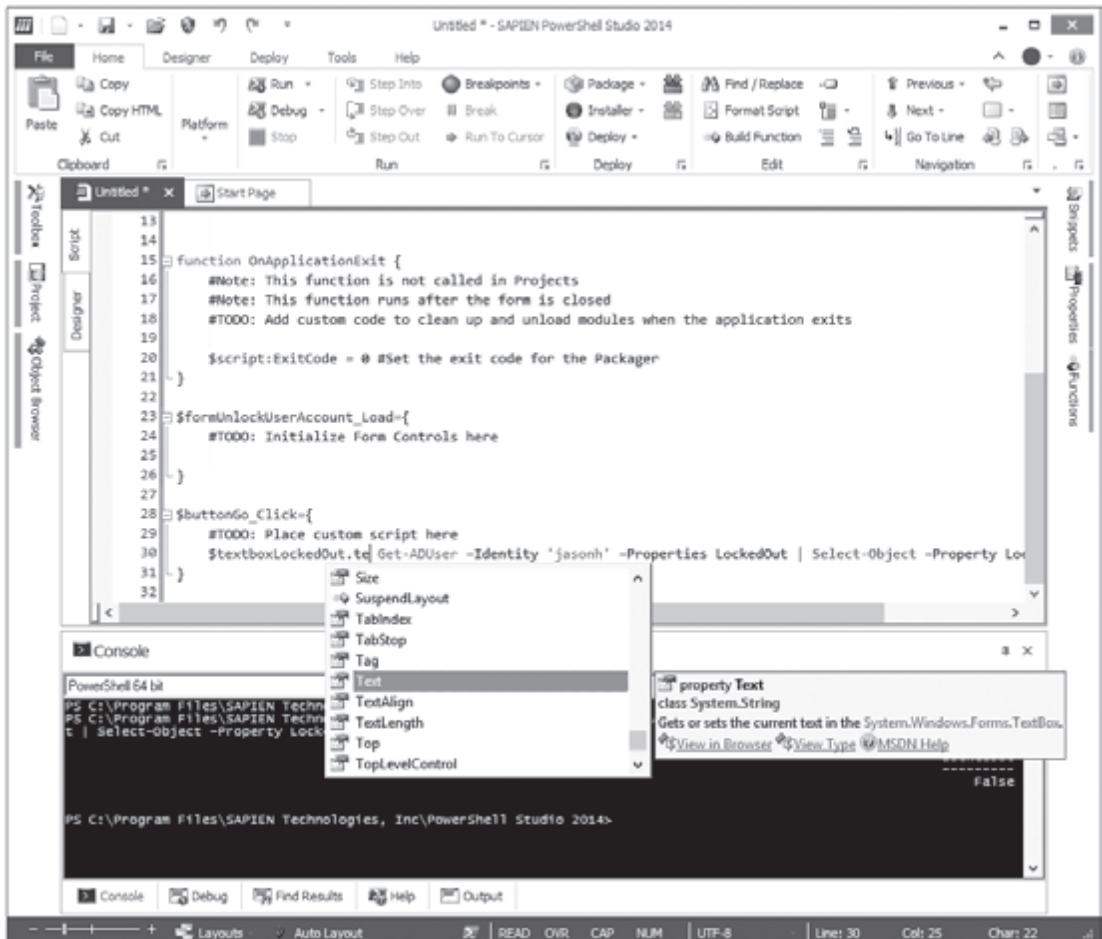
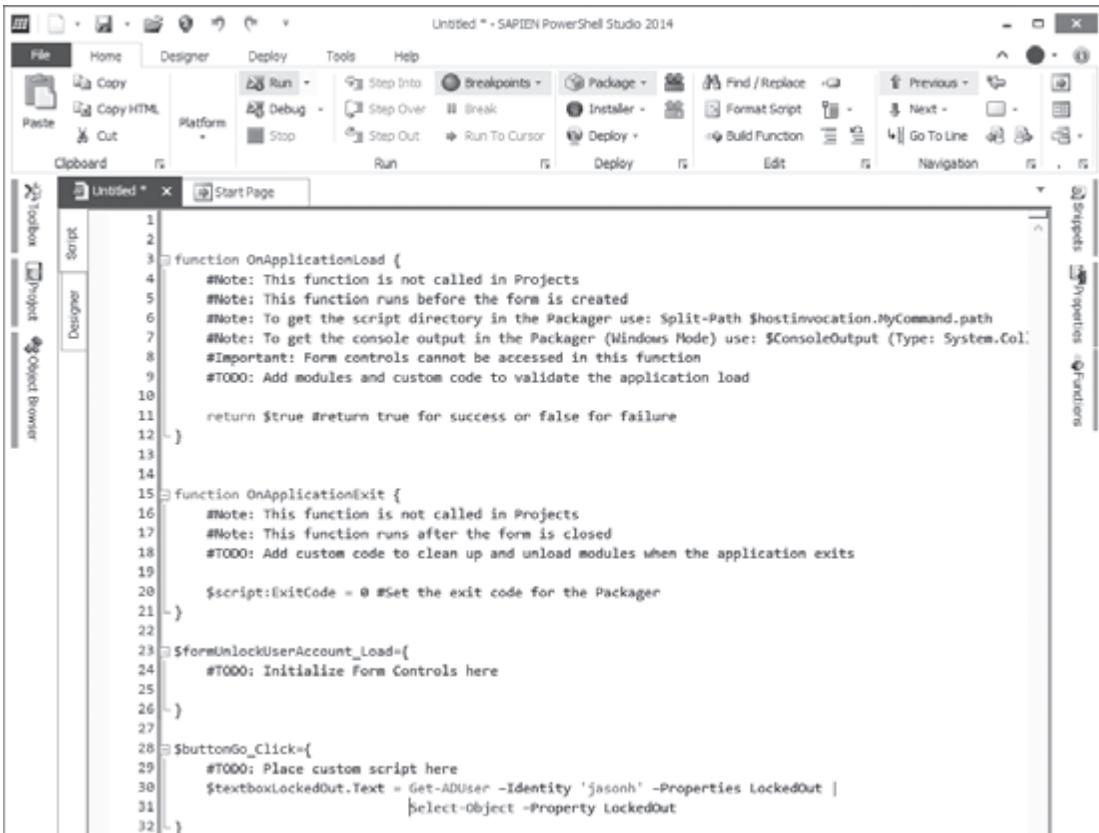


Figure 21-21

With that single line of code, we're going to assign the results of our command to the value of the **Text** property of the **LockedOut** textbox. In other words, the results of the command will be placed in the textbox.

## Windows PowerShell: TFM



The screenshot shows the SAPIEN PowerShell Studio 2014 interface. The main window displays a PowerShell script titled "Untitled". The script contains several functions:

```
function OnApplicationLoad {
    #Note: This function is not called in Projects
    #Note: This function runs before the form is created
    #Note: To get the script directory in the Packager use: Split-Path $hostInvocation.MyCommand.Path
    #Note: To get the console output in the Packager (Windows Mode) use: $ConsoleOutput (Type: System.Collections.Generic.List`1[System.String])
    #Important: Form controls cannot be accessed in this function
    #TODO: Add modules and custom code to validate the application load

    return $true #return true for success or false for failure
}

function OnApplicationExit {
    #Note: This function is not called in Projects
    #Note: This function runs after the form is closed
    #TODO: Add custom code to clean up and unload modules when the application exits

    $script:ExitCode = 0 #Set the exit code for the Packager
}

$formUnlockUserAccount_Load={
    #TODO: Initialize Form Controls here
}

$buttonGo_Click={
    #TODO: Place custom script here
    $textboxLockedOut.Text = Get-ADUser -Identity 'jasonh' -Properties LockedOut |
        Select-Object -Property LockedOut
}
```

Figure 21-22

We select **Run** from the PowerShell Studio ribbon.

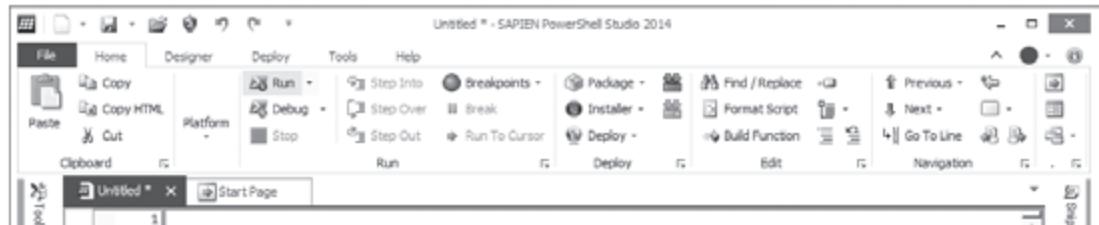


Figure 21-23

We're prompted to save our form, which we will name "UnlockUserAccount".

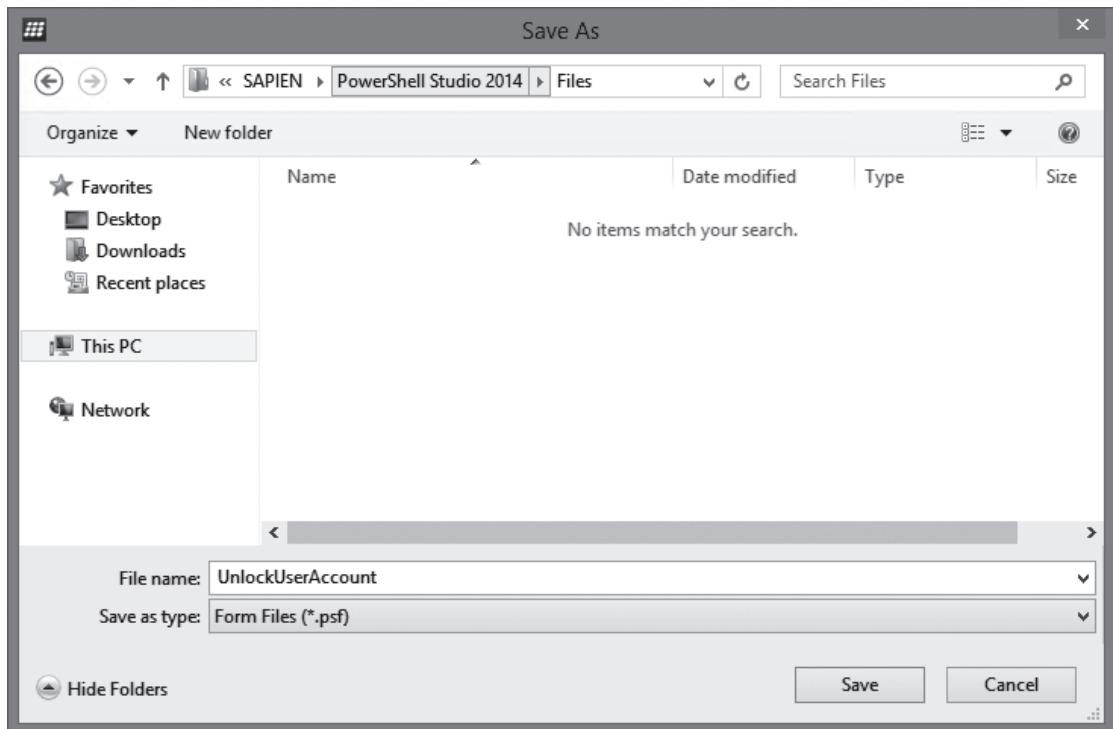


Figure 21-24

We click **Go** and receive unexpected results due to an object being assigned to the **LockedOut** textbox.



Figure 21-25

What we need is for the string value of the **LockedOut** property to be assigned to our textbox, so we make a slight modification to our code and replace the **-Property** parameter with the **-ExpandProperty** parameter.

```
$textboxLockedOut.Text = Get-ADUser -Identity 'jasonh' -Properties LockedOut | Select-Object -ExpandProperty LockedOut
```



```

14
15 function OnApplicationExit {
16     #Note: This function is not called in Projects
17     #Note: This function runs after the form is closed
18     #TODO: Add custom code to clean up and unload modules when the application exits
19
20     $script:ExitCode = 0 #Set the exit code for the Packager
21 }
22
23 $formUnlockUserAccount_Load={
24     #TODO: Initialize Form Controls here
25
26 }
27
28 $buttonGo_Click={
29     #TODO: Place custom script here
30     $textboxLockedOut.Text = Get-ADUser -Identity 'jasonh' -Properties LockedOut |
31             Select-Object -ExpandProperty LockedOut
32 }

```

Figure 21-26

We select **Run File** again and this time the results are what we expected.



Figure 21-27

Now, we'll remove the hardcoded username in the code that we were using to test with and replace it with whatever value that's typed in the **UserName** text box. Whatever user name is typed into that text-box will be used as the username to check for an account lockout.

```

UnlockUserAccount.ps1 - SAPIEN PowerShell Studio 2014

File Home Designer Deploy Tools Help
Copy Copy HTML Cut Clipboard
Run Step Into Breakpoints
Debug Step Over Break
Stop Step Out Run To Cursor
Platform Installer
Run Deploy
Find / Replace Format Script
Build Function
Previous Next Go To Line
Edit Navigation

UnlockUserAccount.ps1 StartPage

14
15 function OnApplicationExit {
16     #Note: This function is not called in Projects
17     #Note: This function runs after the form is closed
18     #TODO: Add custom code to clean up and unload modules when the application exits
19
20     $script:ExitCode = 0 #Set the exit code for the Packager
21 }
22
23 $formUnlockUserAccount_Load={
24     #TODO: Initialize Form Controls here
25
26 }
27
28 $buttonGo_Click={
29     #TODO: Place custom script here
30     $textBoxLockedOut.Text = Get-ADUser -Identity $textBoxUserName.Text -Properties LockedOut |
31         Select-Object -ExpandProperty LockedOut
32 }

```

Figure 21-28

We enter a username, click **Go**, and it works. The problem we see, though, is that a user can modify what's in the **LockedOut** textbox.

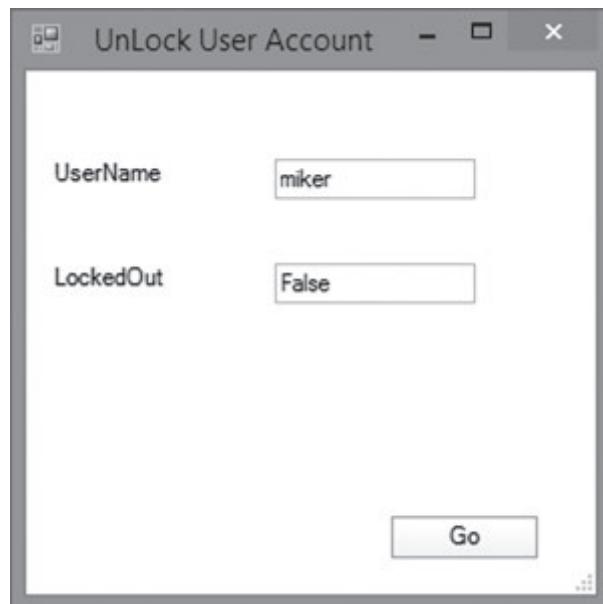


Figure 21-29

We select the **Designer** tab and we'll set the **ReadOnly** property to **True** for that textbox, to prevent a user from being able to modify its value.

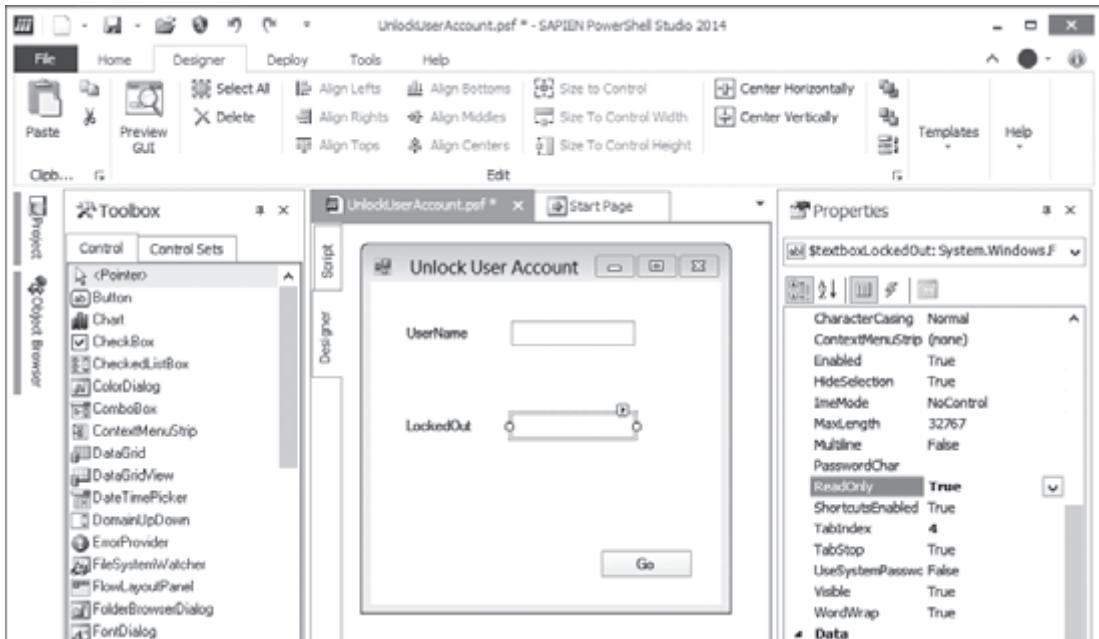


Figure 21-30

Now when our tool is ran, the result displayed in the **LockedOut** textbox is not modifiable.



Figure 21-31

We have a very basic application at this point and it can be used to check to see if a user is locked out or not. However, we want to add the ability to unlock a user if they are locked out. We're going to add a check box to our form and name it "Unlock User". We set the **-Enabled** property to **False**, because we don't want the user of this tool using that particular control until they've checked to see whether the specified user account is locked out or not.

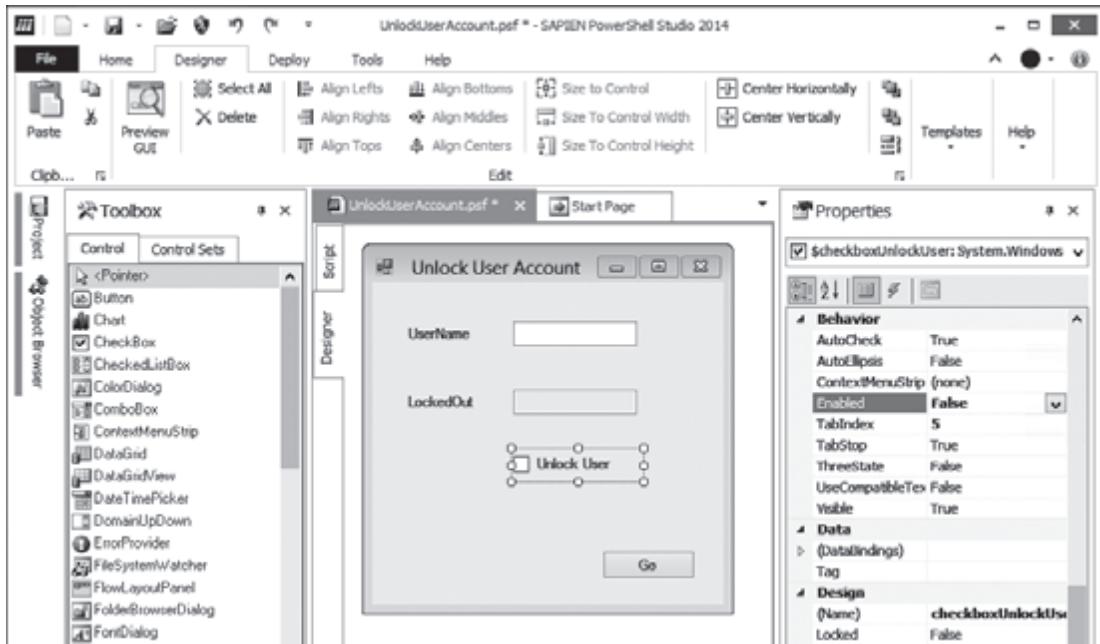


Figure 21-32

We've simply placed an **If** statement inside of our **Go** button click event, to enable the **UnlockUser** check box if the **Text** of **textboxUnlockUser** equals **True**.

When the **LockedOut** textbox contains **False**, the **Unlock User** check box is not enabled.



Figure 21-33

When it contains **True**, the check box is enabled.

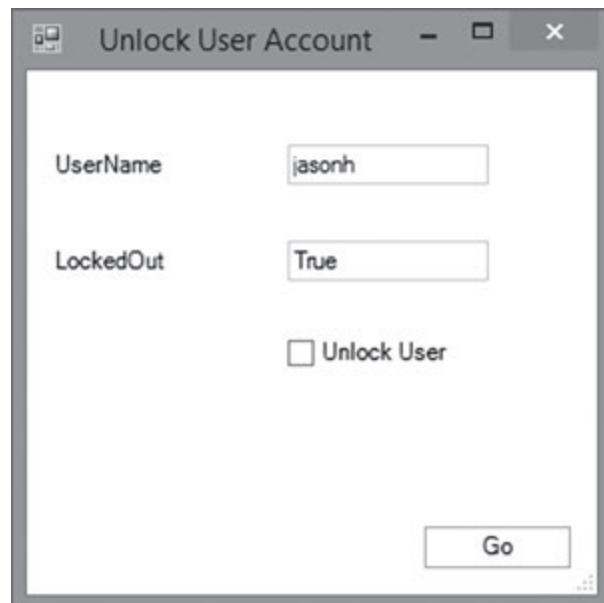
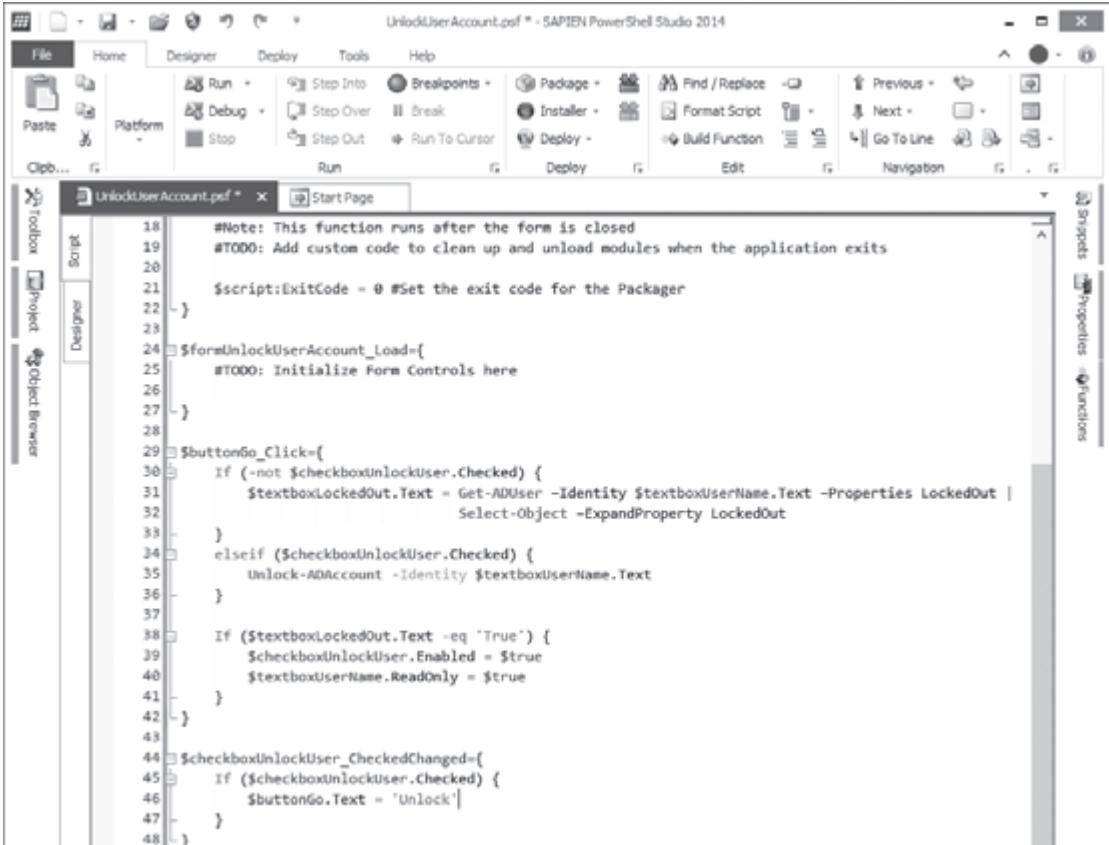


Figure 21-34

Now we just need to add the controls and code so that the user can be unlocked. Instead of adding additional buttons, which may confuse the person using our tool, we'll try to lead them down a path by changing the existing controls as necessary, so there's less that can go wrong.

We've added another **If** statement to our **Go** button click event, which retrieves the user's locked out status if the **UnlockUser** check box is not checked and will unlock the user account if it is checked. We added a line in our previous **If** statement, so the **UserName** textbox is set to **ReadOnly** if the user account is locked out, since we don't want this field changed between the time the status is checked and the account is unlocked. We added the default event handler to our check box by double-clicking on it on the form. We also added code to change the text of our button to "Unlock" if the check box is checked.



```

UnlockUserAccount.ps1 - SAPIEN PowerShell Studio 2014

File Home Designer Deploy Tools Help
Run Step Into Breakpoints Package Find / Replace Previous
Debug Step Over Break Installer Format Script Next
Stop Step Out Run To Cursor Deploy Build Function Go To Line
Run Deploy Edit Navigation
Clipboard Platform
Start Page

UnlockUserAccount.ps1 * Start Page

18     #Note: This function runs after the form is closed
19     #TODO: Add custom code to clean up and unload modules when the application exits
20
21     $script:ExitCode = 0 #Set the exit code for the Packager
22 }
23
24 $formUnlockUserAccount_Load={
25     #TODO: Initialize Form Controls here
26
27 }
28
29 $buttonGo_Click={
30     If (-not $checkboxUnlockUser.Checked) {
31         $textboxLockedOut.Text = Get-ADUser -Identity $textBoxUserName.Text -Properties LockedOut |
32             Select-Object -ExpandProperty LockedOut
33     }
34     elseif ($checkboxUnlockUser.Checked) {
35         Unlock-ADAccount -Identity $textBoxUserName.Text
36     }
37
38     If ($textboxLockedOut.Text -eq 'True') {
39         $checkboxUnlockUser.Enabled = $true
40         $textBoxUserName.ReadOnly = $true
41     }
42 }
43
44 $checkboxUnlockUser_CheckedChanged={
45     If ($checkboxUnlockUser.Checked) {
46         $buttonGo.Text = 'Unlock'
47     }
48 }

```

Figure 21-35

The main issue with our tool now is that the user isn't given any feedback to say whether the locked out account was unlocked or not. We'll add a label to accomplish this.

## Creating WinForm tools with PowerShell Studio

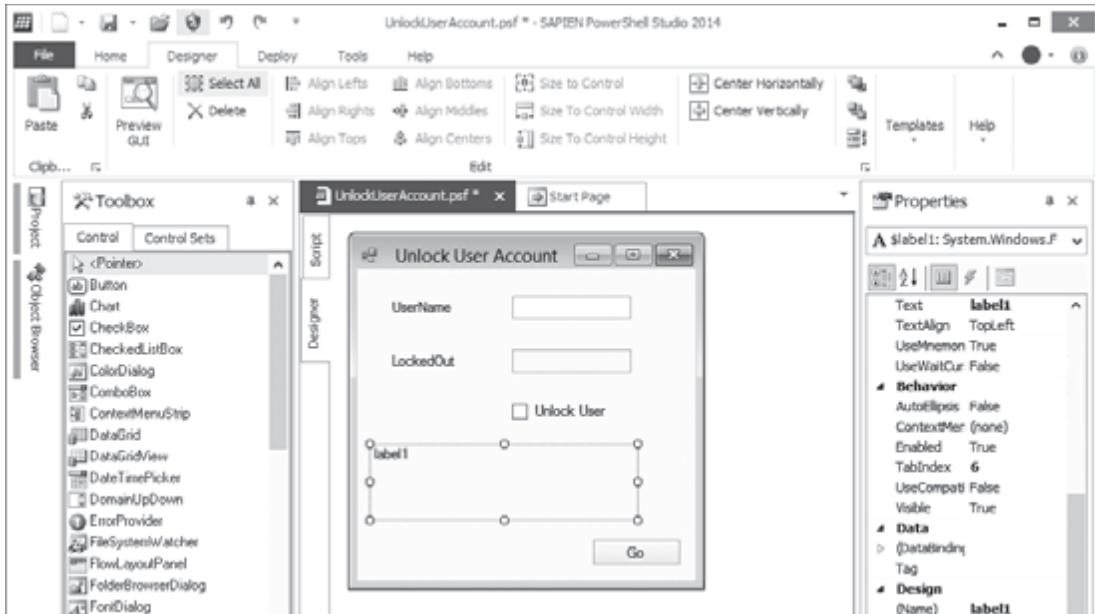


Figure 21-36

We'll remove the text for this label so that it doesn't exist on the form and we'll change the text to display messages to the user of our tool. We've named this label **labelResults**.

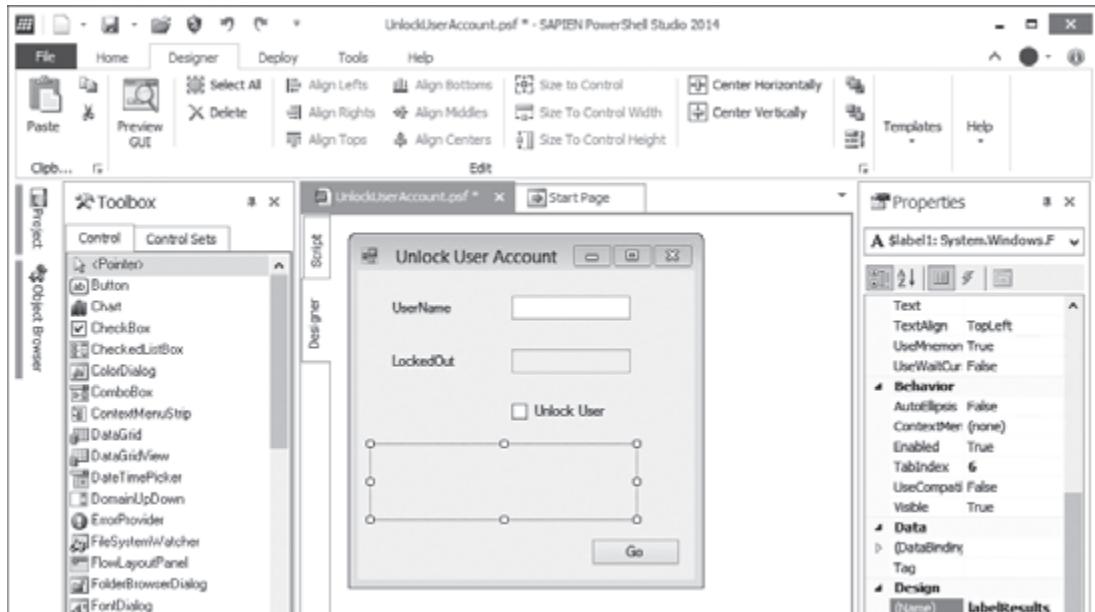


Figure 21-37

## Windows PowerShell: TFM

We've added a line to our code to give the user feedback about the account being unlocked. We've also added error handling and we're using the same label to display the error information.

The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The title bar reads "UnlockUserAccount.ps1 - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The toolbar contains icons for Paste, Platform, Run, Debug, Stop, Step Into, Step Over, Break, Step Out, Run To Cursor, Package, Installer, Deploy, Find / Replace, Format Script, Previous, Next, Go To Line, and Edit. The main window displays the PowerShell script "UnlockUserAccount.ps1". The script contains code for handling button click events and checkbox checked changed events, specifically for unlocking a user account. The code uses Try-Catch blocks to handle errors and updates a label with status information. The script is currently in the "Script" tab of the editor.

```
$buttonGo_Click={  
    If (-not $checkboxUnlockUser.Checked) {  
        try {  
            $textboxLockedOut.Text = Get-ADUser -Identity $textboxUserName.Text -Properties LockedOut  
            Select-Object -ExpandProperty LockedOut  
        }  
        catch {  
            $labelResults.Text = "An Unexpected Error has Occurred"  
        }  
  
        If ($textboxLockedOut.Text -eq 'True') {  
            $checkboxUnlockUser.Enabled = $true  
            $textboxUserName.ReadOnly = $true  
        }  
    }  
    elseif ($checkboxUnlockUser.Checked) {  
        try {  
            Unlock-ADAccount -Identity $textboxUserName.Text  
        }  
        catch {  
            $labelResults.Text = "An Unexpected Error has Occurred"  
        }  
  
        $labelResults.Text = "$($textboxUserName.Text) has been Successfully Unlocked"  
    }  
}  
  
$checkboxUnlockUser_CheckedChanged={  
    If ($checkboxUnlockUser.Checked) {  
        $buttonGo.Text = 'Unlock'  
    }  
}  
}
```

Figure 21-38

When a user is unlocked, the label displays status information to the tool user so that they know that the account was indeed unlocked.



Figure 21-39

We now want a way for our tool user to be able to clear the results and query the status of another user, without having to close and reopen the tool.

We've added logic so the button displays "Clear" after retrieving the locked out status if the account is not locked out or after unlocking an account.

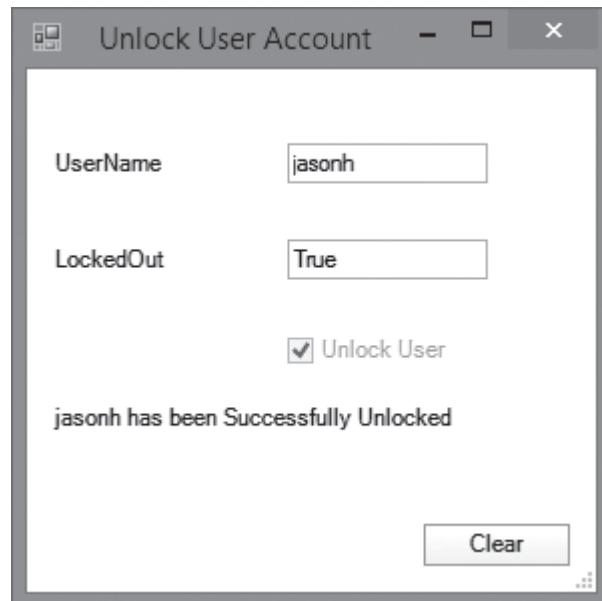


Figure 21-40

We've also added code so that all of the controls are reset to their default settings when the **Clear** button is clicked. If the user of our tool checks and unchecks the checkmark, the button will also change back and forth between **Unlock** and **Clear**.

```
$buttonGo_Click={  
    If (-not ($checkboxUnlockUser.Checked) -and $buttonGo.Text -ne 'Clear') {  
        try {  
            $textboxLockedOut.Text = Get-ADUser -Identity $textboxUserName.Text `  
                -Properties LockedOut -ErrorAction 'Stop' |  
                Select-Object -ExpandProperty LockedOut  
        }  
        catch {  
            $labelResults.Text = "An Unexpected Error has Occurred"  
        }  
  
        $textboxUserName.ReadOnly = $true  
        $buttonGo.Text = 'Clear'  
  
        If ($textboxLockedOut.Text -eq 'True') {  
            $checkboxUnlockUser.Enabled = $true  
        }  
    }  
    elseif ($checkboxUnlockUser.Checked -and $buttonGo.Text -ne 'Clear') {  
        try {  
            Unlock-ADAccount -Identity $textboxUserName.Text -ErrorAction 'Stop'  
        }  
        catch {  
            $labelResults.Text = "An Unexpected Error has Occurred"  
        }  
        finally {  
            $buttonGo.Text = 'Clear'  
        }  
  
        $labelResults.Text = "$(($textboxUserName.text) has been Successfully Unlocked)"  
        $checkboxUnlockUser.Enabled = $false  
    }  
    elseif ($buttonGo.Text -eq 'Clear') {  
        $checkboxUnlockUser.Checked = $false  
        $checkboxUnlockUser.Enabled = $false  
        $labelResults.Text = ''  
        $textboxLockedOut.Text = ''  
        $textboxUserName.Text = ''  
        $textboxUserName.ReadOnly = $false  
        $buttonGo.Text = 'Go'  
    }  
}  
  
$checkboxUnlockUser_CheckedChanged={  
    If ($checkboxUnlockUser.Checked) {  
        $buttonGo.Text = 'Unlock'  
    }  
    else {  
        $buttonGo.Text = 'Clear'  
    }  
}
```

Our **Go** button click event handler has become fairly complicated at this point.

The screenshot shows the SAPiEN PowerShell Studio 2014 application window. The title bar reads "UnlockUserAccount.ps1 - SAPiEN PowerShell Studio 2014". The menu bar includes File, Home, Designer, Deploy, Tools, and Help. The main area is a code editor with the file "UnlockUserAccount.ps1" open. The code is a PowerShell script with several event handlers defined. The left sidebar shows tabs for Script, Designer, and Deploy. The bottom navigation bar includes Console, Debug, Find Results, Help, Output, Layouts, Auto Layout, and various status indicators like READ, OVR, CAP, NUM, and UTF-8.

```

29 $buttonGo_Click = {
30     If (-not ($checkboxUnlockUser.Checked) -and $buttonGo.Text -ne 'Clear') {
31         try {
32             $textboxLockedOut.Text = Get-ADUser -Identity $textboxUserName.Text -Properties LockedOut
33                         Select-Object -ExpandProperty LockedOut
34         }
35         catch {
36             $labelResults.Text = "An Unexpected Error has Occurred"
37         }
38
39         $textboxUserName.ReadOnly = $true
40         $buttonGo.Text = 'Clear'
41
42         If ($textboxLockedOut.Text -eq 'True') {
43             $checkboxUnlockUser.Enabled = $true
44         }
45     }
46     elseif ($checkboxUnlockUser.Checked -and $buttonGo.Text -ne 'Clear') {
47         try {
48             Unlock-ADAccount -Identity $textboxUserName.Text -ErrorAction 'Stop'
49         }
50         catch {
51             $labelResults.Text = "An Unexpected Error has Occurred"
52         }
53         finally {
54             $buttonGo.Text = 'Clear'
55         }
56
57         $labelResults.Text = "$($textboxUserName.text) has been Successfully Unlocked"
58         $checkboxUnlockUser.Enabled = $false
59     }
60     elseif ($buttonGo.Text -eq 'Clear') {
61         $checkboxUnlockUser.Checked = $false
62         $checkboxUnlockUser.Enabled = $false
63         $labelResults.Text = ''
64         $textboxLockedOut.Text = ''
65         $textboxUserName.Text = ''
66         $textboxUserName.ReadOnly = $false
67         $buttonGo.Text = 'Go'
68     }
69 }
70
71 $checkboxUnlockUser_CheckedChanged = {
72     If ($checkboxUnlockUser.Checked) {
73         $buttonGo.Text = 'Unlock'
74     }
75     else {
76         $buttonGo.Text = 'Clear'
77     }
78 }

```

Figure 21-41

The only other event handler we've defined is **CheckChanged** for our check box.

### Windows PowerShell: TFM

```
71 $checkboxUnlockUser_CheckedChanged={  
72     If ($checkboxUnlockUser.Checked) {  
73         $buttonGo.Text = 'Unlock'  
74     }  
75     else {  
76         $buttonGo.Text = 'Clear'  
77     }  
78 }
```

Figure 21-42

It's possible to package our tool as an executable by going to the **Deploy** menu and selecting **Build**.

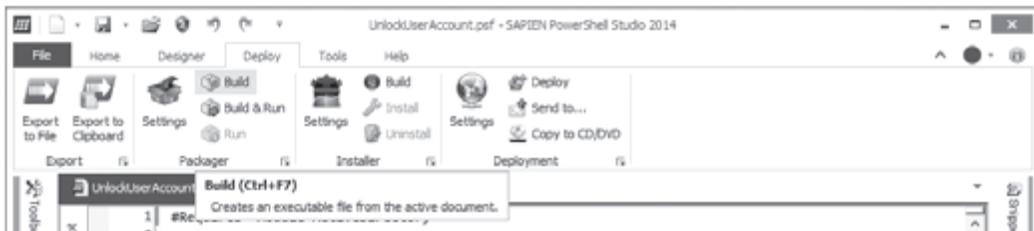


Figure 21-43

It's also possible to include alternate credentials, in case you are providing this tool to users who don't have access to unlock user accounts.

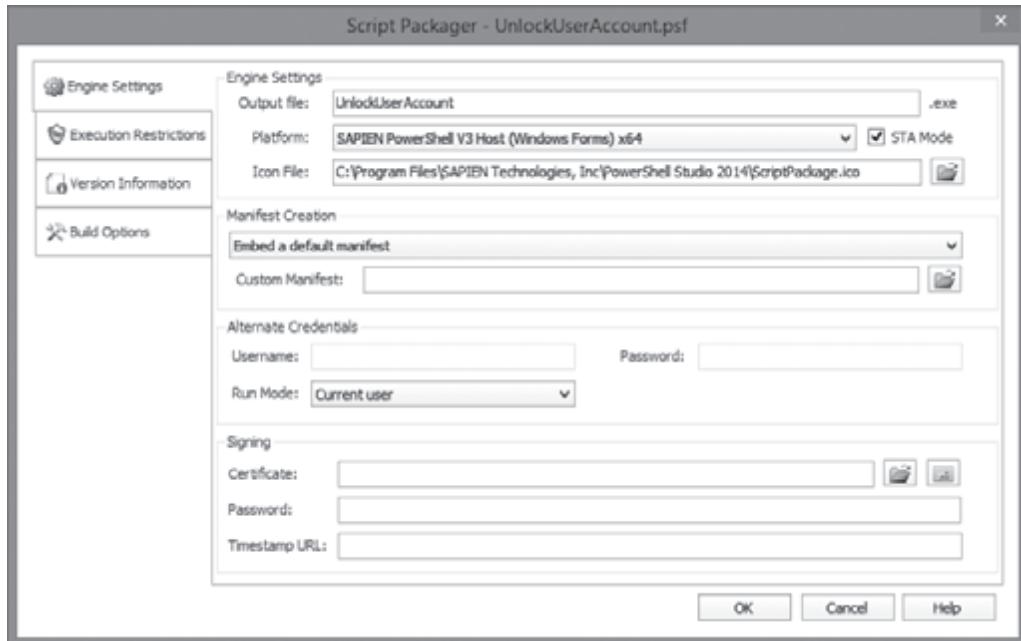


Figure 21-44

As you can see, we now have an executable tool which the users can run just as they would any other application.

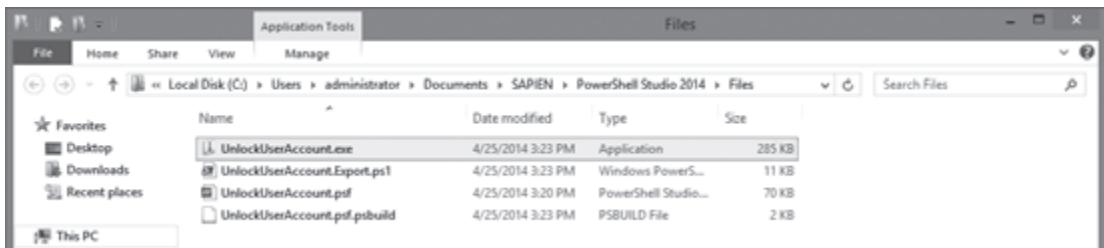


Figure 21-45

---

**Note:**

Although our application or tool is now an executable, this does not remove the requirement that PowerShell be installed on the machine that it is being run on.

We now have a fully functional tool that can be used to check to see if a user is locked out or not, and unlock an account if it is. As you have seen in this chapter, it's fairly simple to create complex GUI tools or applications with PowerShell Studio and a little knowledge of PowerShell.

## Exercise 21 — Create a WinForm tool

The Help desk staff at your company has heard about the tool to retrieve manufacturer, model, and serial number information from remote computers. They would like you to add a graphical user interface to this tool so they can also use it.

### Task 1

Using the information that you've gained about retrieving the manufacturer, model, and serial number information of a computer during the exercises in the last few chapters, use PowerShell to build a GUI to retrieve and display this information for the Help desk personnel.

## Chapter 22

# Desired state configuration

### What is desired state configuration?

Desired State Configuration (DSC) is a new feature in PowerShell version 4. It's an extension to PowerShell, designed to place servers and/or workstations in your infrastructure in a specific state, based on a declarative configuration script created in PowerShell. This configuration script creates a Managed Object Format (MOF) file, which is what the actual state of the server or workstation is modeled to.

The DSC PowerShell cmdlets are contained in a new module named **PSDesiredStateConfiguration**, as shown in Figure 22-1.

```
Get-Command -Module PSDesiredStateConfiguration
```

CommandType	Name	ModuleName
Function	Configuration	PSDesiredStateConfiguration
Function	Get-DscConfiguration	PSDesiredStateConfiguration
Function	Get-DscLocalConfigurationManager	PSDesiredStateConfiguration
Function	Get-DscResource	PSDesiredStateConfiguration
Function	New-DSCChecksum	PSDesiredStateConfiguration
Function	Restore-DscConfiguration	PSDesiredStateConfiguration
Function	Test-DscConfiguration	PSDesiredStateConfiguration
Cmdlet	Set-DscLocalConfigurationManager	PSDesiredStateConfiguration
Cmdlet	Start-DscConfiguration	PSDesiredStateConfiguration

Figure 22-1

## Configuration

Configuration is a new keyword in PowerShell version 4 and it allows you to easily create MOF files. These files are used to define the configuration that you want to apply to the machines in your infrastructure. The configuration itself is defined like a function, where the name is declared after the configuration keyword, followed by a script block.

```
configuration iSCSI {  
}  
}
```

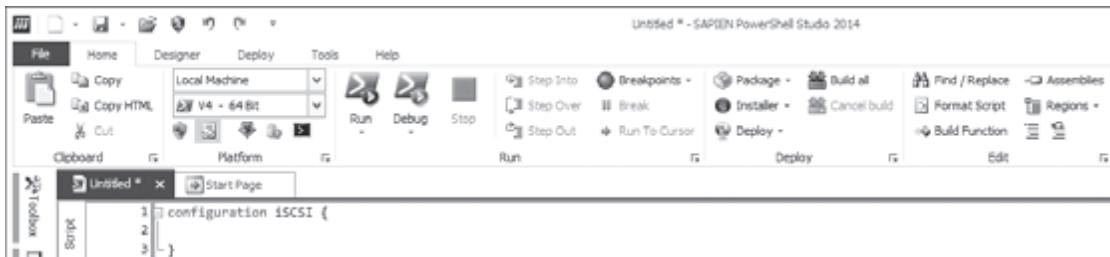


Figure 22-2

Zero or more nodes are defined inside of the configuration script block. If zero nodes exist, a directory named “iSCSI” will be created in the current path, but no MOF file will be created.

We'll define a node for our server, named SQL01.

```
configuration iSCSI {  
    node SQL01 {  
    }  
}
```

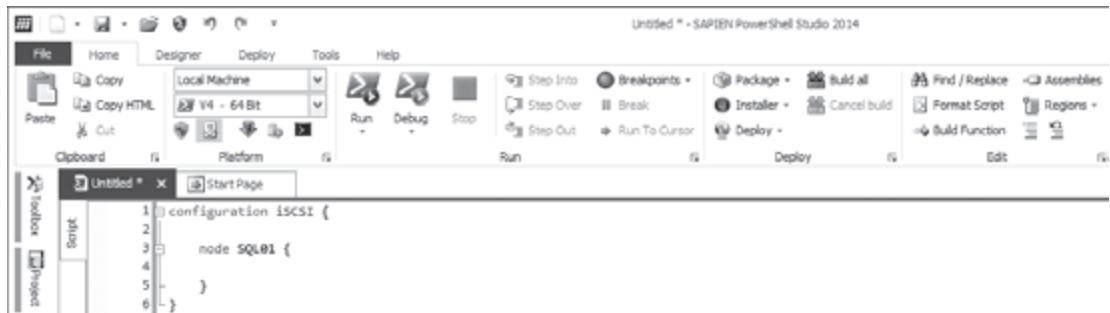


Figure 22-3

If we ran our configuration at this point, a MOF file still wouldn't be created, since we haven't defined any resources yet. Resources? What are resources?

## Resources

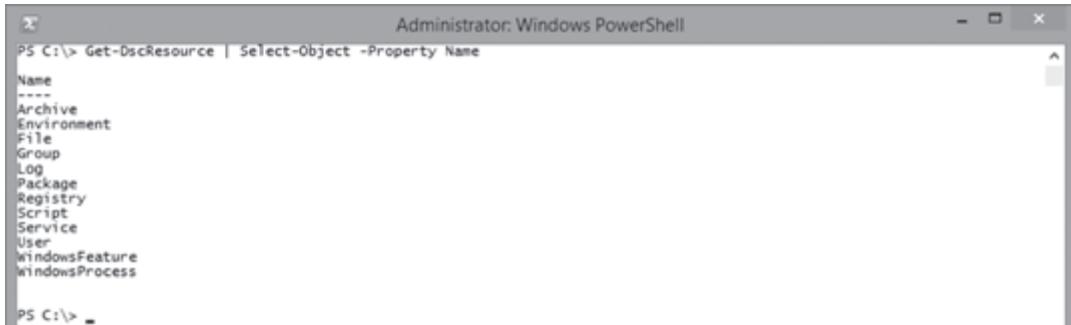
The simplest way to describe resources in DSC is that they are the building blocks that allow us to control our desired configuration preferences. They are the items we use to define “This is what I want the state” for a particular service to be in, for example.

Here is a list of the built-in DSC resources in Windows Server 2012 R2 and Windows 8.1:

- Archive
- Environment
- File
- Group
- Log
- Package
- Process
- Registry
- Role
- Script
- Service
- User

You can always use PowerShell to find DSC resources, as well.

```
Get-DscResource | Select-Object -Property Name
```



```
Administrator: Windows PowerShell
PS C:\> Get-DscResource | Select-Object -Property Name
Name
-----
Archive
Environment
File
Group
Log
Package
Registry
Script
Service
User
WindowsFeature
WindowsProcess
```

Figure 22-4

Note that the command shown in Figure 22-4 will search all modules that are explicitly loaded or in your **\$env:PSModulePath** for resources, so if any custom resources exist, they will also be listed. Keep in mind that it can take a considerable amount of time to return all of the resources, depending on how many modules you have loaded. We'll add the **-Verbose** parameter to our previous command, to give you an idea of what happens behind the scenes.

## Windows PowerShell: TFM

```
Get-DSCResource -Verbose | Select-Object -Property Name
```

```
Administrator: Windows PowerShell
PS C:\> Get-DSCResource -Verbose | Select-Object -Property Name
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Appx\Appx.psm1'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\AssignedAccess\AssignedAccess.psm1'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BestPractices\Microsoft.BestPractices.Cmdlets.dll'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BitLocker\Microsoft.BitLocker.Structures.dll'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BitLocker\BitLocker.psdi'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BitLocker\Microsoft.BitLocker.Structures.dll'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BitLocker\BitLocker.psdi'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCacheClientSettingData.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCacheContentServerSettingData.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCacheHostedCacheServerSettingData.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCacheNetworkSettingData.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCacheOrchestrator.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCachePrimaryPublicationCacheFile.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCachePrimaryReplicationCacheFile.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCacheSecondaryReplicationCacheFile.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\BranchCache\BranchCacheStatus.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\ClusterAwareUpdating\ClusterAwareUpdating.dll'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Defender\MSFT_MpComputerStatus.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Defender\MSFT_MpPreference.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Defender\MSFT_NpThreat.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Defender\MSFT_NpThreatCatalog.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Defender\MSFT_NpThreatDetection.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Defender\MSFT_NpScan.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Defender\MSFT_NpSignature.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\DFSN\MSFT_DfsNamespace.DfsNamespace.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\DFSN\MSFT_DfsNamespaceFolder.DfsNamespaceFolder.cdxm'.
VERBOSE: Loading module from path 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\DFSN\MSFT_DfsNamespaceAccess.DfsNamespaceAccess.cdxm'.
VERBOSE: Loading module from path
```

Figure 22-5

We're going to be adding a Windows Feature to our SQL01 server with DSC, so we're specifically looking to see what our options are for the **WindowsFeature** DSC resource.

```
(Get-DSCResource -Name WindowsFeature).properties | Format-Table -AutoSize
```

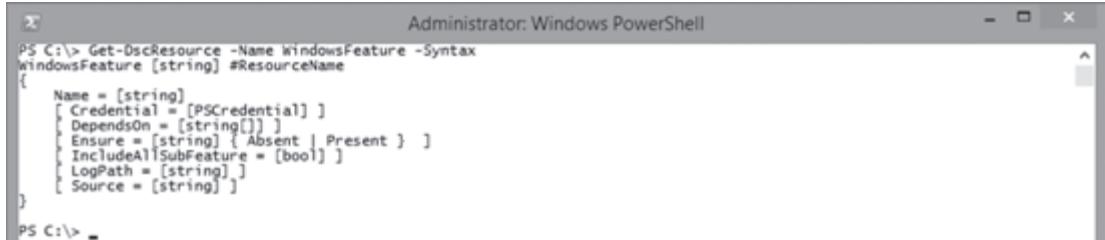
Name	PropertyType	IsMandatory	Values
Name	[string]	True	{}
Credential	[PSCredential]	False	{}
DependsOn	[string[]]	False	{}
Ensure	[string]	False	{Absent, Present}
IncludeAllSubFeature	[bool]	False	{}
LogPath	[string]	False	{}
Source	[string]	False	{}

Figure 22-6

#### Desired state configuration

We can also see what the syntax is for the **WindowsFeature** DSC resource from within PowerShell.

```
Get-DscResource -Name WindowsFeature -Syntax
```

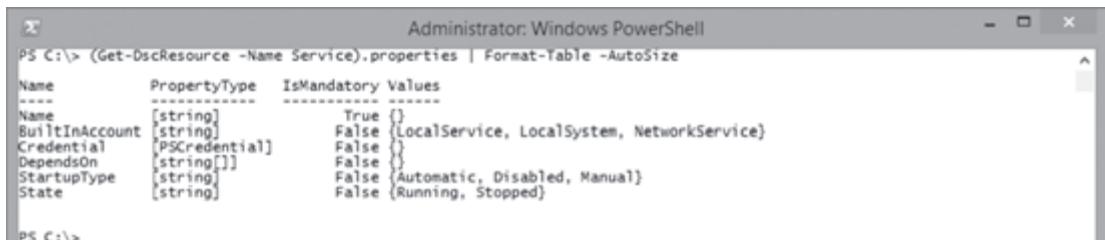


```
Administrator: Windows PowerShell
PS C:\> Get-DscResource -Name WindowsFeature -Syntax
WindowsFeature [string] #ResourceName
{
    Name = [string]
    [ Credential = [PSCredential] ]
    DependsOn = [String[]]
    Ensure = [string] { Absent | Present }
    IncludeAllSubFeature = [bool]
    LogPath = [string]
    Source = [string]
}
PS C:\> _
```

Figure 22-7

We'll be configuring a service as well, so we'll check to see what our options are for that particular DSC resource. Notice that the options are different between the two different DSC resources that we'll be using.

```
(Get-DscResource -Name Service).properties | Format-Table -AutoSize
```

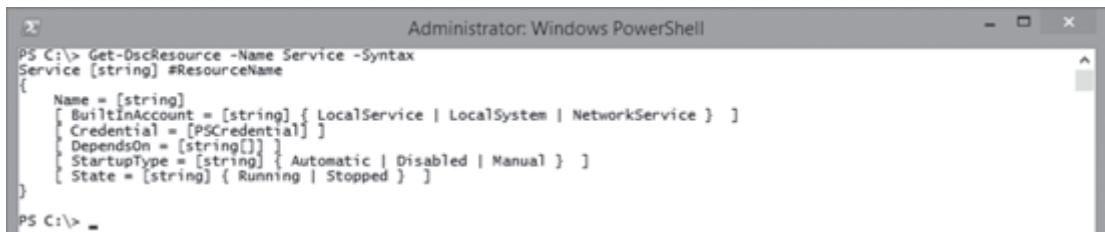


```
Administrator: Windows PowerShell
PS C:\> (Get-DscResource -Name Service).properties | Format-Table -AutoSize
Name      PropertyType  IsMandatory Values
----      -----        -----   -----
Name      [string]      True     {}
BuiltInAccount [string]  False    {LocalService, LocalSystem, NetworkService}
Credential  [PSCredential]  False   {}
DependsOn   [string[]]   False   {}
StartupType  [string]    False   {Automatic, Disabled, Manual}
State       [string]    False   {Running, Stopped}
PS C:\> _
```

Figure 22-8

We'll also check the syntax for the DSC service resource.

```
Get-DscResource -Name Service -Syntax
```



```
Administrator: Windows PowerShell
PS C:\> Get-DscResource -Name Service -Syntax
Service [string] #ResourceName
{
    Name = [string]
    [ BuiltInAccount = [string] { LocalService | LocalSystem | NetworkService } ]
    Credential = [PSCredential]
    DependsOn = [String[]]
    StartupType = [string] { Automatic | Disabled | Manual }
    [ State = [string] { Running | Stopped } ]
}
PS C:\> _
```

Figure 22-9

Many online resources for DSC state that **-Requires** is an available property for all DSC resources, but that must have been in the pre-RTM releases. It appears, based on the available options for both of these resources, that the property name for that particular option is **-DependsOn**.

We'll add a total of two DSC resources to our configuration. We want to make sure that the **MultiPath-IO** feature is present; the **MSiSCSI** service is set to start automatically and is running.

```
configuration iSCSI {
    node SQL01 {
        WindowsFeature MultipathIO {
            Name = 'Multipath-IO'
            Ensure = 'Present'
        }

        Service iSCSIService {
            Name = 'MSiSCSI'
            StartupType = 'Automatic'
            State = 'Running'
        }
    }
}
```



Figure 22-10

Notice that the syntax of our configuration is a little different than others you may have previously seen or others you may see in the future. We prefer formatting our code this way, because it seems to be more IT professional-centric, instead of being too developer-oriented. We also prefer to have the name of the item we're referencing, such as **Name = 'Multipath-IO'** (as shown in the **WindowsFeature** resource section of Figure 22-10), defined before any other resource properties. Doing this means it's easy to tell what we're working with and at least with our two different DSC resources, **-Name** is the only mandatory property.

## Configuration reusability

We believe that if you're going to do something more than once, script it. However, we also believe that if you're going to script it, script it once and make it reusable, because there's really no sense in even using PowerShell if you're only going to write the same code over and over again, like repeating a task over and over again in the GUI.

To make our configuration script reusable, we've made a few changes. We've added a parameter block, just like you would in a function. We even added parameter validation, which through our testing appears to work properly, although there's no official documentation on whether or not parameter validation is supported in a configuration script. Instead of a static node name, we can now specify a node name via the **-ComputerName** parameter, when calling our configuration script.

```
configuration iSCSI {
    param (
        [Parameter(Mandatory=$True)]
        [string[]]$ComputerName
    )

    node $ComputerName {
        WindowsFeature MultipathIO {
            Name = 'Multipath-IO'
            Ensure = 'Present'
        }

        Service iSCSIService {
            Name = 'MSiSCSI'
            StartupType = 'Automatic'
            State = 'Running'
        }
    }
}
```

## DSC configuration MOF file

We've added "iSCSI", the name of our configuration script, to the bottom of our code, to call the configuration similar to calling a function.

## Windows PowerShell: TFM



The screenshot shows the SAPiEN PowerShell Studio 2014 interface. The main window displays a PowerShell script titled 'Untitled' with the following content:

```
1 configuration iSCSI {
2
3     param (
4         [Parameter(Mandatory=$True)]
5         [string[]]$ComputerName
6     )
7
8     node $ComputerName {
9
10        WindowsFeature MultipathIO {
11            Name = "Multipath-IO"
12            Ensure = "Present"
13        }
14
15        Service iSCSIService {
16            Name = 'MSiSCSI'
17            StartupType = "Automatic"
18            State = "Running"
19        }
20
21    }
22
23 }
24 iSCSI
```

Figure 22-11

We select the code shown in Figure 22-11, and then we ran our code by pressing Ctrl + Shift + F8. As shown in Figure 22-12, we were prompted for computer names because we made that parameter mandatory. We provide the name **SQL01** and then **Server01**, just to show how our configuration can be used to created multiple MOF files for different nodes.



The screenshot shows a PowerShell console window. The command run is:

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

The output shows the cmdlet prompting for computer names:

```
iSCSI at command pipeline position 1
Supply values for the following parameters:
ComputerName[0]: SQL01
ComputerName[1]: Server01
ComputerName[2]:
```

Then it lists the directory and files created:

```
directory: C:\iSCSI

Mode LastWriteTime Length Name
---- ---- - - -
-a-- 10/2/2013 12:08 PM 1642 SQL01.mof
-a-- 10/2/2013 12:08 PM 1648 Server01.mof
```

Figure 22-12

We'll take a peek at the contents of our newly created MOF files just to see what kind of magic pixie dust they contain.

```
Get-Content -Path C:\iSCSI\SQL01.mof
```

Desired state configuration

Administrator: Windows PowerShell

```
PS C:\> Get-Content -Path C:\iSCSI\SQL01.mof
/*
@TargetNode="SQL01"
@GeneratedBy="administrator"
@GenerationDate="10/02/2013 11:27:24"
@GenerationHost="PC01"
*/
instance of MSFT_RoleResource as $MSFT_RoleResource1ref
{
    ResourceID = "[WindowsFeature]MultipathIO";
    Ensure = "Present";
    SourceInfo = "::3::9::WindowsFeature";
    Name = "Multipath-IO";
    ModuleName = "MSFT_RoleResource";
    ModuleVersion = "1.0";
};

instance of MSFT_ServiceResource as $MSFT_ServiceResource1ref
{
    ResourceID = "[Service]iSCSIService";
    State = "Running";
    SourceInfo = "::7::9::Service";
    Name = "MSiSCSI";
    StartupType = "Automatic";
    ModuleName = "MSFT_ServiceResource";
    ModuleVersion = "1.0";
};

instance of OMI_ConfigurationDocument
{
    Version="1.0.0";
    Author="administrator";
    GenerationDate="10/02/2013 11:27:24";
    GenerationHost="PC01";
};

PS C:\> =
```

Figure 22-13

```
Get-Content -Path C:\iSCSI\Server01.mof
```

Administrator: Windows PowerShell

```
PS C:\> Get-Content -Path C:\iSCSI\Server01.mof
/*
@TargetNode="Server01"
@GeneratedBy="administrator"
@GenerationDate="10/02/2013 12:03:14"
@GenerationHost="PC01"
*/
instance of MSFT_RoleResource as $MSFT_RoleResource1ref
{
    ResourceID = "[WindowsFeature]MultipathIO";
    Ensure = "Present";
    SourceInfo = "::7::9::WindowsFeature";
    Name = "Multipath-IO";
    ModuleName = "MSFT_RoleResource";
    ModuleVersion = "1.0";
};

instance of MSFT_ServiceResource as $MSFT_ServiceResource1ref
{
    ResourceID = "[Service]iSCSIService";
    State = "Running";
    SourceInfo = "::11::9::Service";
    Name = "MSiSCSI";
    StartupType = "Automatic";
    ModuleName = "MSFT_ServiceResource";
    ModuleVersion = "1.0";
};

instance of OMI_ConfigurationDocument
{
    Version="1.0.0";
    Author="administrator";
    GenerationDate="10/02/2013 12:03:14";
    GenerationHost="PC01";
};

PS C:\> =
```

Figure 22-14

## Deployment

We'll first check to see if the role that we're working with is already installed and what state the iSCSI service is in, before continuing.

```
Invoke-Command -ComputerName SQL01 {Get-WindowsFeature -Name Multipath-IO}
```

```
Administrator: Windows PowerShell
PS C:\> Invoke-Command -ComputerName SQL01 {Get-WindowsFeature -Name Multipath-IO}

PSComputerName      : SQL01
RunspaceId          : $7789700-798b-4cac-aeb-a67ae3c6b2ce
Name                : Multipath-IO
DisplayName         : Multipath I/O
Description         : Multipath I/O, along with the Microsoft Device Specific Module (DSM) or a third-party DSM, provides support for using multiple data paths to a storage device on Windows.
Installed           : False
InstallState        : Available
FeatureType         : Feature
Path                : Multipath I/O
Depth               : 1
DependsOn          : {}
Parent              : ServerComponent_Multipath_IO
ServerComponentDescriptor : ServerComponent_Multipath_IO
SubFeatures         : {}
SystemService       : {}
Notification        : {}
BestPracticesModelId : {}
EventQuery          : {}
PostConfigurationNeeded : False
AdditionalInfo      : {MajorVersion, InstallName, MinorVersion, NumericId}

PS C:\> _
```

Figure 22-15

```
Invoke-Command -ComputerName SQL01 {Get-WMIObject -Class Win32_Service -Filter "Name = 'MSiSCSI'")}
```

```
Administrator: Windows PowerShell
PS C:\> Invoke-Command -ComputerName SQL01 {Get-WMIObject -Class Win32_Service -Filter "Name = 'MSiSCSI'"}

ExitCode      : 1077
Name          : MSiSCSI
ProcessId    : 0
StartMode    : Manual
State         : Stopped
Status        : OK
PSComputerName : SQL01

PS C:\> _
```

Figure 22-16

As you can see, the role isn't currently enabled, the services exist—but its startup mode and state are both currently different than what we want.

We'll now apply our DSC configuration MOF files that we just created to SQL01 and Server01. By default, a PowerShell job would be created. However, specifying the **-Wait** parameter runs the command interactively. We also specified the **-Verbose** parameter, so you can see the details of what's going on while the command is running. The verbose output isn't necessarily readable, but it should look familiar.

### Desired state configuration

```
Start-DscConfiguration -ComputerName SQL01, Server01 -Path iSCSI -Wait -Verbose
```

The screenshot shows an Administrator Windows PowerShell window. The command PS C:\> Start-DscConfiguration -ComputerName SQL01, Server01 -Path iSCSI -Wait -Verbose is being run. The output displays verbose logs from the DSC configuration process. It includes messages about performing operations like 'Invoke CimMethod' with specific parameters, sending configurations to local managers, and starting resources. It also shows the installation of the Multipath-Io feature on the servers. A warning message at the end of the log suggests enabling automatic updates for better management.

```
Administrator: Windows PowerShell
PS C:\> Start-DscConfiguration -ComputerName SQL01, Server01 -Path iSCSI -Wait -Verbose
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, 'methodName' = 'SendConfigurationApply', 'className' = 'MSFT_DSCLocalConfigurationManager', 'namespaceName' = 'root/Microsoft/Windows/DesiredStateConfiguration'.
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, 'methodName' = 'SendConfigurationApply', 'className' = 'MSFT_DSCLocalConfigurationManager', 'namespaceName' = 'root/Microsoft/Windows/DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer PC01 with user sid 5-1-5-21-419186775-871882884-2772554839-500.
VERBOSE: [SQL01]: LCM: [ Start Set ]
VERBOSE: An LCM method call arrived from computer PC01 with user sid 5-1-5-21-419186775-871882884-2772554839-500.
VERBOSE: [SQL01]: LCM: [ Start Set ]
VERBOSE: [SQL01]: LCM: [ Start Resource ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ Start Test ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] The operation 'Get-WindowsFeature' started: Multipath-Io
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] The operation 'Get-WindowsFeature' succeeded: Multipath-Io
VERBOSE: [SQL01]: LCM: [ End Test ] [WindowsFeature]MultipathIO] in 0.5620 seconds.
VERBOSE: [SQL01]: LCM: [ Skip Set ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ End Resource ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ Start Resource ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ Start Test ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ End Test ] [Service]iSCSIService] in 0.0630 seconds.
VERBOSE: [SQL01]: LCM: [ Skip Set ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ End Resource ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ End Set ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ End Set ] in 0.9/00 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: [SQL01]: LCM: [ Start Resource ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ Start Test ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] The operation 'Get-WindowsFeature' started: Multipath-Io
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] The operation 'Get-WindowsFeature' succeeded: Multipath-Io
VERBOSE: [SQL01]: LCM: [ End Test ] [WindowsFeature]MultipathIO] in 7.5740 seconds.
VERBOSE: [SQL01]: LCM: [ Start Set ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] Installation started...
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] Prerequisite processing started...
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] Prerequisite processing succeeded.
WARNING: [SQL01]: [WindowsFeature]MultipathIO] Windows automatic updating is not enabled.
To ensure that your newly-installed role or feature is automatically updated, turn on Windows Update.
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] Installation succeeded.
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] successfully installed the feature.
VERBOSE: Multipath-Io
VERBOSE: [SQL01]: LCM: [ End Set ] [WindowsFeature]MultipathIO] in 8.3490 seconds.
VERBOSE: [SQL01]: LCM: [ End Resource ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ Start Resource ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ Start Test ] [Service]iSCSIService]
VERBOSE: [SQL01]: [Service]iSCSIService] Startup type for service 'MSiSCSI' is 'Manual'. It does not match 'Automatic'.
VERBOSE: [SQL01]: LCM: [ End Test ] [Service]iSCSIService] in 0.1410 seconds.
VERBOSE: [SQL01]: LCM: [ Start Set ] [Service]iSCSIService]
VERBOSE: [SQL01]: [Service]iSCSIService] Service 'MSiSCSI' started.
VERBOSE: [SQL01]: LCM: [ End Set ] [Service]iSCSIService] in 1.9500 seconds.
VERBOSE: [SQL01]: LCM: [ End Resource ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ End set ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ End Set ] in 19.1830 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 19.5 seconds
PS C:\>
```

Figure 22-17

We're going to go ahead and create CimSessions to our SQL01 and Server01 servers.

```
$CimSession = New-CimSession -ComputerName SQL01, Server01
```

The screenshot shows an Administrator Windows PowerShell window. The command PS C:\> \$CimSession = New-CimSession -ComputerName SQL01, Server01 is being run. The output shows the creation of a new CimSession object named \$CimSession, which represents the connection to the specified computers.

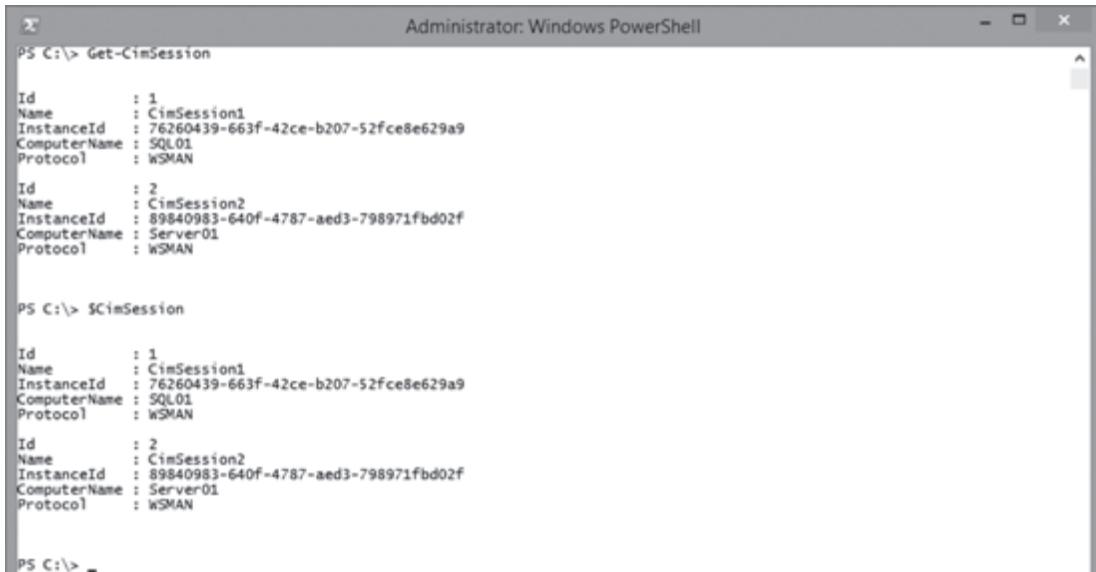
```
Administrator: Windows PowerShell
PS C:\> $CimSession = New-CimSession -ComputerName SQL01, Server01
PS C:\>
```

Figure 22-18

## Windows PowerShell: TFM

You may think the command in Figure 22-18 creates a single CimSession to both servers, but it's just an easy way to create multiple CimSessions in one line of code and store all of them in a single variable. As you can see in Figure 22-19, there are two CimSessions.

```
Get-CimSession  
$CimSession
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The session starts with the command "Get-CimSession", which outputs two objects representing CimSessions. Each object has properties: Id, Name, InstanceId, ComputerName, and Protocol. Both sessions have an Id of 1 and 2 respectively, and are named "CimSession1" and "CimSession2". Their InstanceIds are unique GUIDs, and they are connected to the same computer names ("SQL01" and "Server01") via the WSMAN protocol. The session then displays the variable "\$CimSession", which contains the same two objects. Finally, the user types a command starting with "PS C:\> \_" at the prompt.

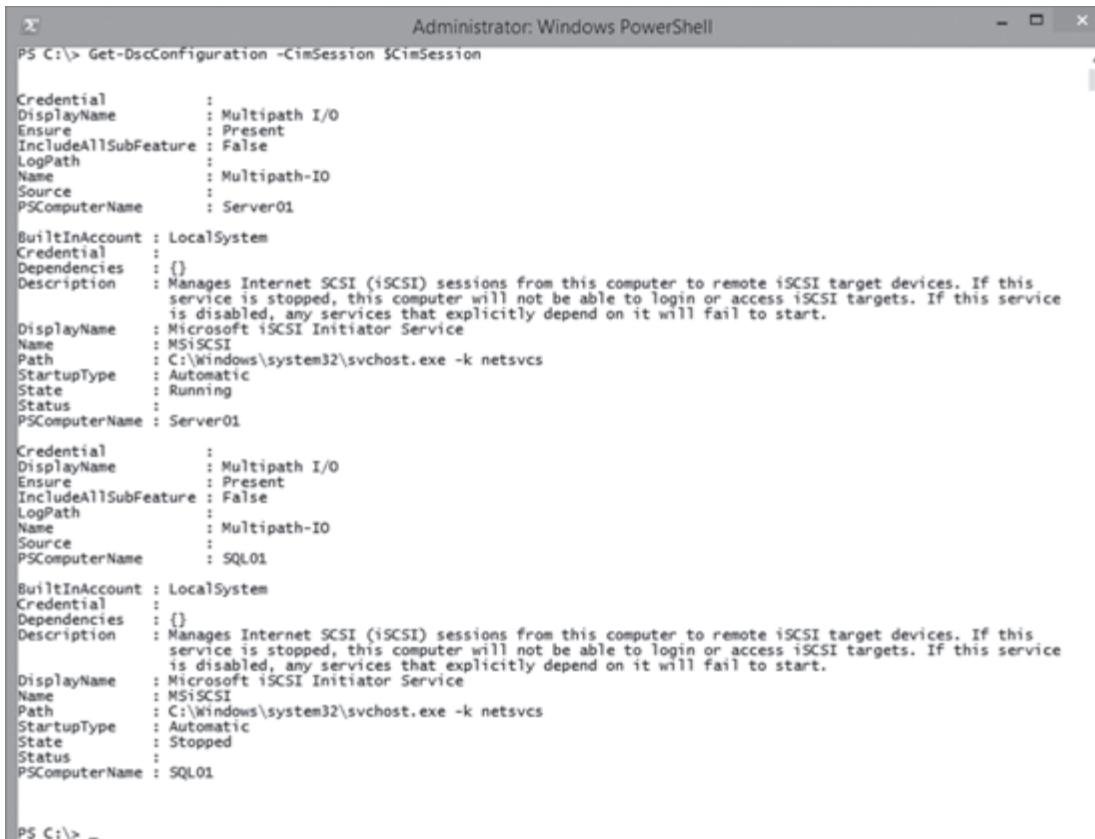
```
PS C:\> Get-CimSession  
  
Id      : 1  
Name    : CimSession1  
InstanceId : 76260439-663f-42ce-b207-52fce8e629a9  
ComputerName : SQL01  
Protocol   : WSMAN  
  
Id      : 2  
Name    : CimSession2  
InstanceId : 89840983-640f-4787-aed3-798971fbd02f  
ComputerName : Server01  
Protocol   : WSMAN  
  
PS C:\> $CimSession  
  
Id      : 1  
Name    : CimSession1  
InstanceId : 76260439-663f-42ce-b207-52fce8e629a9  
ComputerName : SQL01  
Protocol   : WSMAN  
  
Id      : 2  
Name    : CimSession2  
InstanceId : 89840983-640f-4787-aed3-798971fbd02f  
ComputerName : Server01  
Protocol   : WSMAN  
  
PS C:\> _
```

Figure 22-19

There are several DSC cmdlets that have a **-CimSession** parameter for working with remote computers, but not a **-ComputerName** parameter. We'll use the CimSessions that we created in Figure 22-19, with the **Get-DscConfiguration** cmdlet, which retrieves the current configuration of a node or nodes.

```
Get-DscConfiguration -CimSession $CimSession
```

### Desired state configuration



```
Administrator: Windows PowerShell
PS C:\> Get-DscConfiguration -CimSession $CimSession

Credential          : Multipath I/O
Ensure              : Present
IncludeAllSubFeature : False
LogPath             :
Name                : Multipath-Io
Source              :
PSCoputerName      : Server01

BuiltInAccount      : LocalSystem
Credential          :
Dependencies        : {}
Description         : Manages Internet SCSI (iSCSI) sessions from this computer to remote iSCSI target devices. If this service is stopped, this computer will not be able to login or access iSCSI targets. If this service is disabled, any services that explicitly depend on it will fail to start.
DisplayName         : Microsoft iSCSI Initiator Service
Name                : MSiSCSI
Path                : C:\Windows\system32\svchost.exe -k netsvcs
StartupType         : Automatic
State               : Running
Status              :
PSCoputerName      : Server01

Credential          : Multipath I/O
Ensure              : Present
IncludeAllSubFeature : False
LogPath             :
Name                : Multipath-Io
Source              :
PSCoputerName      : SQL01

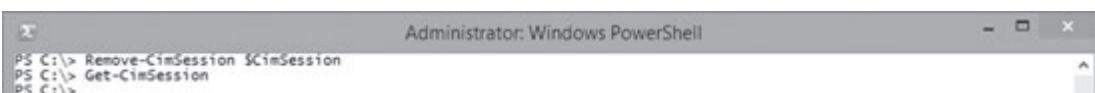
BuiltInAccount      : LocalSystem
Credential          :
Dependencies        : {}
Description         : Manages Internet SCSI (iSCSI) sessions from this computer to remote iSCSI target devices. If this service is stopped, this computer will not be able to login or access iSCSI targets. If this service is disabled, any services that explicitly depend on it will fail to start.
DisplayName         : Microsoft iSCSI Initiator Service
Name                : MSiSCSI
Path                : C:\Windows\system32\svchost.exe -k netsvcs
StartupType         : Automatic
State               : Stopped
Status              :
PSCoputerName      : SQL01

PS C:\> _
```

Figure 22-20

We're now going to remove our CimSessions.

```
Remove-CimSession $CimSession
Get-CimSession
```



```
Administrator: Windows PowerShell
PS C:\> Remove-CimSession $CimSession
PS C:\> Get-CimSession
PS C:\> _
```

Figure 22-21

Why did we remove our CimSessions? We're going to show you something we discovered, so that you'll have as much practical knowledge as possible by the time you complete this chapter.

Notice that **Test-DscConfiguration** doesn't have a computer name parameter, only a **-CimSession** parameter, for working with remote computers.

## Windows PowerShell: TFM

```
Get-Command -Name Test-DscConfiguration | Select-Object -ExpandProperty Parameters
```

Key	Value
---	-----
CimSession	System.Management.Automation.ParameterMetadata
ThrottleLimit	System.Management.Automation.ParameterMetadata
AsJob	System.Management.Automation.ParameterMetadata
Verbose	System.Management.Automation.ParameterMetadata
Debug	System.Management.Automation.ParameterMetadata
ErrorAction	System.Management.Automation.ParameterMetadata
WarningAction	System.Management.Automation.ParameterMetadata
ErrorVariable	System.Management.Automation.ParameterMetadata
WarningVariable	System.Management.Automation.ParameterMetadata
OutVariable	System.Management.Automation.ParameterMetadata
OutBuffer	System.Management.Automation.ParameterMetadata
PipelineVariable	System.Management.Automation.ParameterMetadata

Figure 22-22

We can, however, specify a remote computer name by using the **-CimSession** parameter, without creating a CimSession or doing anything else to communicate with a remote computer. This won't necessarily work with all cmdlets, but it does work with all of the DSC cmdlets.

```
Test-DscConfiguration -CimSession SQL01
```

```
Test-DscConfiguration -CimSession Server01
```

```
Administrator: Windows PowerShell
PS C:\> Test-DscConfiguration -CimSession SQL01
False
PS C:\> Test-DscConfiguration -CimSession Server01
True
PS C:\>
```

Figure 22-23

We could have used the **Test-DscConfiguration** cmdlet to test both computers in one line of code. However in Figure 22-23, we chose to run the command twice so that it would be easier to distinguish which computer had experienced some sort of configuration drift. As we can see, our server named SQL01 has had some sort of configuration drift and while we could use the **-Verbose** parameter to see the details, those details don't really matter to us—unless this is something that regularly happens and we need to determine the cause of the configuration drift.

At this point, all we want to do is restore our server back into its desired configuration, so we'll just reapply our DSC configuration.

```
Start-DscConfiguration -ComputerName SQL01 -Path iSCSI -Wait -Verbose
```

### Desired state configuration

```
Administrator: Windows PowerShell
PS C:\> Start-DscConfiguration -ComputerName SQL01 -Path iSCSI -Wait -Verbose
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters: "methodName =
SendConfigurationApply,'ClassName' = MSFT_DSCLocalConfigurationManager', namespaceName' =
root\Microsoft\Windows\DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer PC01 with user sid S-1-5-21-419186775-871882884-2772554839-500.
VERBOSE: [SQL01]: LCM: [ Start Set ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ Start Resource ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ Start Test ] [WindowsFeature]MultipathIO] The operation 'Get-WindowsFeature' started:
Multipath-IO
VERBOSE: [SQL01]: [WindowsFeature]MultipathIO] The operation 'Get-WindowsFeature'.
succeeded: Multipath-IO
VERBOSE: [SQL01]: LCM: [ End Test ] [WindowsFeature]MultipathIO] in 0.5630 seconds.
VERBOSE: [SQL01]: LCM: [ Skip Set ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ End Resource ] [WindowsFeature]MultipathIO]
VERBOSE: [SQL01]: LCM: [ Start Resource ] [Service]iSCSIService
VERBOSE: [SQL01]: LCM: [ Start Test ] [Service]iSCSIService] in 0.0470 seconds.
VERBOSE: [SQL01]: LCM: [ End Test ] [Service]iSCSIService]
VERBOSE: [SQL01]: LCM: [ Start Set ] [Service]iSCSIService]
VERBOSE: [SQL01]: [ End Set ] [Service]iSCSIService] Service 'MSiSCSI' started.
VERBOSE: [SQL01]: [ End Set ] [Service]iSCSIService] in 1.6040 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 2.651 seconds.
PS C:\>
```

Figure 22-24

We'll now check the SQL01 server again to verify it's in the desired configuration.

```
Test-DscConfiguration -CimSession SQL01
```

```
Administrator: Windows PowerShell
PS C:\> Test-DscConfiguration -CimSession SQL01
True
PS C:\>
```

Figure 22-25

There's also a **Restore-DscConfiguration** cmdlet, but what that cmdlet does is restore the previous DSC configuration on a node.

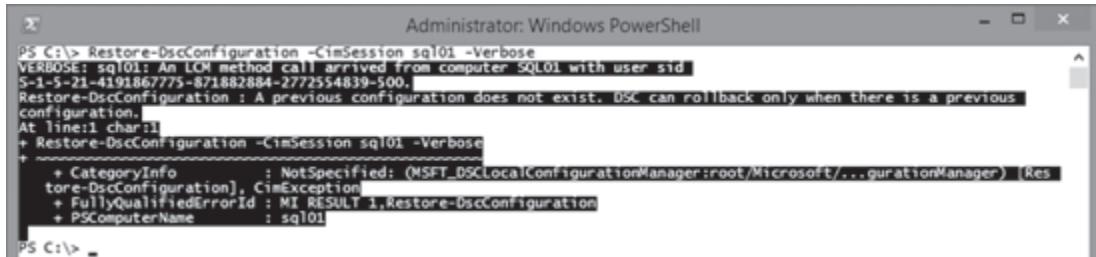
```
Restore-DscConfiguration -CimSession SQL01 -Verbose
```

```
Administrator: Windows PowerShell
PS C:\> Restore-DscConfiguration -CimSession sql01 -Verbose
VERBOSE: sql01: An LCM method call arrived from computer SQL01 with user sid
S-1-5-21-419186775-871882884-2772554839-500.
VERBOSE: sql01: [LCM: [ Start Rollback ]
VERBOSE: sql01: [LCM: [ Start Resource ] [WindowsFeature]MultipathIO]
VERBOSE: sql01: [LCM: [ Start Test ] [WindowsFeature]MultipathIO] The operation 'Get-WindowsFeature'
started: Multipath-IO
VERBOSE: sql01: [WindowsFeature]MultipathIO] The operation 'Get-WindowsFeature'.
succeeded: Multipath-IO
VERBOSE: sql01: LCM: [ End Test ] [WindowsFeature]MultipathIO] in 0.3280 seconds.
VERBOSE: sql01: LCM: [ Skip Set ] [WindowsFeature]MultipathIO]
VERBOSE: sql01: LCM: [ End Resource ] [WindowsFeature]MultipathIO]
VERBOSE: sql01: LCM: [ Start Resource ] [Service]iSCSIService
VERBOSE: sql01: LCM: [ Start Test ] [Service]iSCSIService] in 0.0000 seconds.
VERBOSE: sql01: LCM: [ End Test ] [Service]iSCSIService]
VERBOSE: sql01: LCM: [ Skip Set ] [Service]iSCSIService]
VERBOSE: sql01: LCM: [ End Resource ] [Service]iSCSIService] in 0.3910 seconds.
VERBOSE: sql01: LCM: [ End Rollback ] [Service]iSCSIService]
```

Figure 22-26

If only one DSC configuration was ever applied to a node, then there is no prior DSC configuration to restore to, and you will receive an error when attempting to use this cmdlet.

```
Restore-DscConfiguration -CimSession SQL01 -Verbose
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Restore-DscConfiguration -CimSession SQL01 -Verbose". The output shows an error message: "VERBOSE: SQL01: An LCM method call arrived from computer SQL01 with user SID S-1-5-21-419186775-871882884-2772554839-500. Restore-DscConfiguration : A previous configuration does not exist. DSC can rollback only when there is a previous configuration." The command is then repeated: "+ Restore-DscConfiguration -CimSession SQL01 -Verbose". The output for this second command includes a detailed error object with properties: CategoryInfo, CimException, FullyQualifiedErrorId, and PSCo

Figure 22-27

There are two different models for DSC configuration. The Push model is what we've been demonstrating in this chapter. There is also a Pull model, which is where nodes can pull their configuration from a file or web server. The Pull model requires additional configuration and is outside of the scope of this chapter.

## **Exercise 22 – Desired state configuration**

### **Task 1**

Create a Desired State Configuration script to install the Telnet Client on the machines in your test lab environment.

### **Task 2**

Create an MOF file from the configuration script that was created in Task 1, for each of the machines in your test lab environment.

### **Task 3**

Apply the desired state (deploy the MOF files) created in Task 2 to the machines in your test environment.

### **Task 4**

Verify the Telnet Client was successfully installed on all of the machines in your test lab environment.



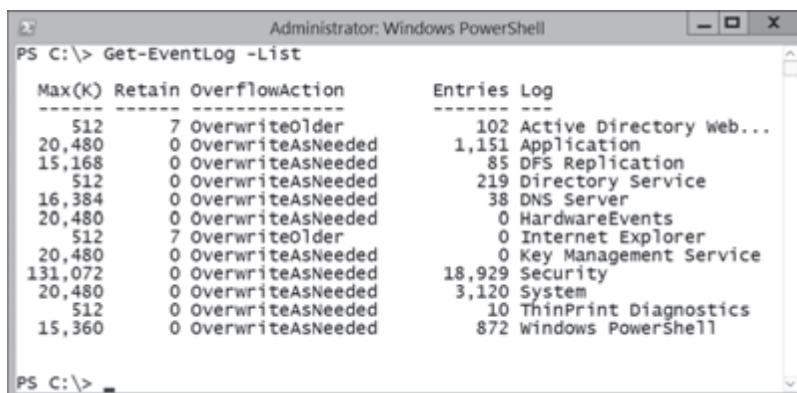
## In Depth 01

# Managing event logs

### Managing event logs

PowerShell has a terrific cmdlet called **Get-EventLog** that makes it easy to find information in a system's event log. Since different systems may have different event logs, one of the first commands you'll want to use is the following:

```
Get-EventLog -List
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-EventLog -List" is run at the prompt. The output displays the configuration of various event logs, including their maximum size (Max(K)), retention period (Retain), and overflow action (OverflowAction). To the right, a list of logs is shown with their respective entry counts.

Max(K)	Retain	OverflowAction	Entries Log
512	7	OverwriteOlder	102 Active Directory Web...
20,480	0	OverwriteAsNeeded	1,151 Application
15,168	0	OverwriteAsNeeded	85 DFS Replication
512	0	OverwriteAsNeeded	219 Directory Service
16,384	0	OverwriteAsNeeded	38 DNS Server
20,480	0	OverwriteAsNeeded	0 HardwareEvents
512	7	OverwriteOlder	0 Internet Explorer
20,480	0	OverwriteAsNeeded	0 Key Management Service
131,072	0	OverwriteAsNeeded	18,929 Security
20,480	0	OverwriteAsNeeded	3,120 System
512	0	OverwriteAsNeeded	10 ThinPrint Diagnostics
15,360	0	OverwriteAsNeeded	872 Windows PowerShell

Figure 1-1

If you run something like the following script, every entry in the log will scroll by.

```
Get-EventLog -LogName "Windows Powershell"
```

Index	Time	EntryType	Source	InstanceID
872	Jul 02 12:28	Information	PowerShell	403
871	Jul 02 12:28	Information	PowerShell	403
870	Jul 02 12:27	Information	PowerShell	400
869	Jul 02 12:27	Information	PowerShell	600
868	Jul 02 12:27	Information	PowerShell	600
867	Jul 02 12:27	Information	PowerShell	600
866	Jul 02 12:27	Information	PowerShell	600
865	Jul 02 12:27	Information	PowerShell	600
864	Jul 02 12:27	Information	PowerShell	600
863	Jul 02 12:27	Information	PowerShell	400
862	Jul 02 12:27	Information	PowerShell	600
861	Jul 02 12:27	Information	PowerShell	600
860	Jul 02 12:27	Information	PowerShell	600
859	Jul 02 12:27	Information	PowerShell	600
858	Jul 02 12:27	Information	PowerShell	600

Figure 1-2

That's probably not very practical, unless you're dumping the contents to another file.

Fortunately, the cmdlet has the parameter **-Newest**, which will display the most recent number of log entries that you specify.

```
Get-EventLog -LogName "Windows PowerShell" -Newest 5
```

Index	Time	EntryType	Source	InstanceID
872	Jul 02 12:28	Information	PowerShell	403
871	Jul 02 12:28	Information	PowerShell	403
870	Jul 02 12:27	Information	PowerShell	400
869	Jul 02 12:27	Information	PowerShell	600
868	Jul 02 12:27	Information	PowerShell	600

Figure 1-3

The default table format usually ends up truncating the event message. If that happens, you can try something like the following:

```
Get-EventLog "Windows PowerShell" -Newest 5 | Format-List
```

Here is an alternative.

```
Get-EventLog -LogName "Windows PowerShell" -Newest 5 |  
Select EntryType, TimeGenerated, EventID, Message |  
Format-List
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Get-EventLog -LogName "Windows PowerShell" -Newest 5 | Select  
EntryType ,TimeGenerated, EventID,Message | Format-List
```

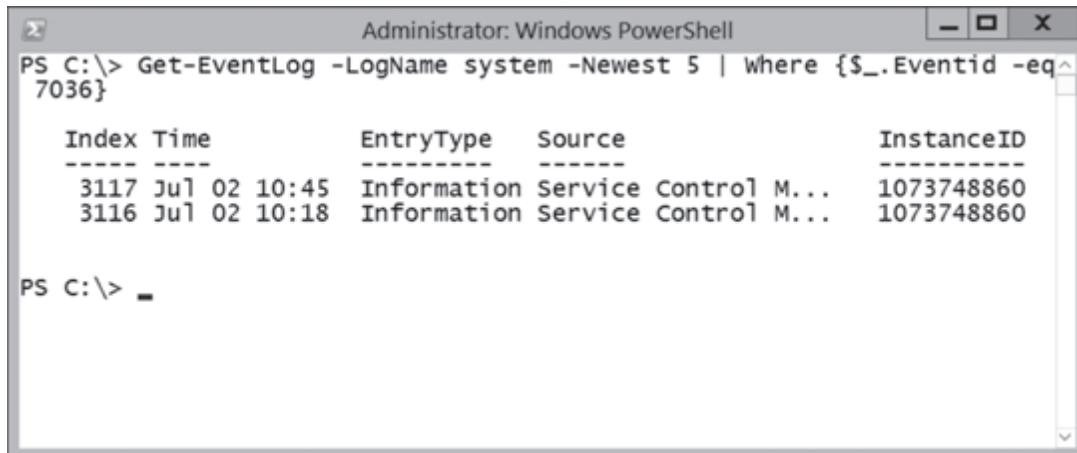
The output displays the following event details:

```
EntryType      : Information  
TimeGenerated  : 7/2/2013 12:28:05 PM  
EventID        : 403  
Message        : Engine state is changed from Available to Stopped.  
Details:  
    NewEngineState=Stopped  
    PreviousEngineState=Available  
SequenceNumber =15  
HostName=Windows Powershell ISE Host
```

Figure 1-4

We've truncated the output, but you get the idea. If you're interested in a specific event ID, use the **Where-Object** cmdlet. Here, we're looking for event log entries with an **EventID** of "7036".

```
Get-EventLog -LogName system -Newest 5 | Where {$_.EventID -eq 7036}
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command run is "Get-EventLog -LogName system -Newest 5 | Where {\$\_.Eventid -eq 7036}". The output shows two events from the System log:

Index	Time	EntryType	Source	InstanceID
3117	Jul 02 10:45	Information	Service Control M...	1073748860
3116	Jul 02 10:18	Information	Service Control M...	1073748860

Figure 1-5

The **Get-EventLog** cmdlet offers some additional capabilities for filtering the events you see, such as the **-Index** parameter, to find a specific event by its index number.

```
Get-EventLog System -Index 108990
```

Use the **-EntryType** parameter to find just specific types of entries, such as **SuccessAudit** entries from the Security log. Valid parameter values are **-Error**, **-Information**, **-FailureAudit**, **-SuccessAudit**, and **-Warning**.

```
Get-EventLog Security -EntryType "SuccessAudit"
```

Use the **-Source** parameter to find entries just from specific sources, such as DCOM.

```
Get-EventLog System -Source DCOM
```

Use the **-List** parameter (by itself) to find a list of available logs.

```
Get-EventLog -List
```

To see that same list as a string, rather than a set of objects, do the following:

```
Get-EventLog -List -AsString
```

Use the **-Before** and **-After** parameters to filter by date.

```
Get-EventLog System -Before 3/1/2014 -After 2/1/2014 -EntryType Error
```

The previous expression will return all **Error** events in the System event log that occurred between 2/1/2014 and 3/1/2014.

Use the **-Message** parameter to find events that contain a particular string. Wildcards are permitted.

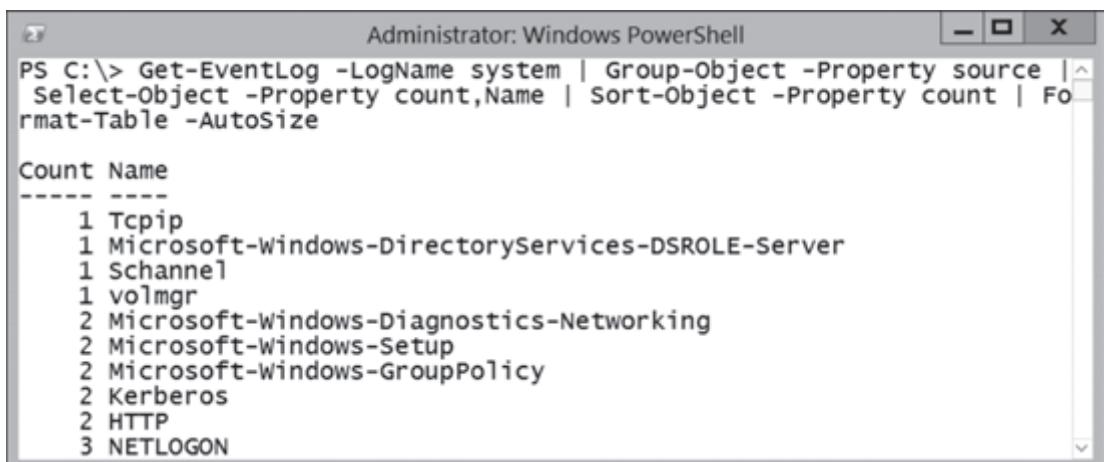
```
Get-EventLog System -Message "*spooler*" -EntryType Error
```

Use the **-Username** parameter to find events for a specific user. Not all events will populate this field.

```
Get-EventLog Application -Username mycompany\administrator
```

Curious about where all your errors are coming from? Try something like the following:

```
Get-EventLog -LogName system | Group-Object -Property source | Select-Object -Property count, Name  
| sort-Object -Property count | Format-Table -AutoSize
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Get-EventLog -LogName system | Group-Object -Property source |  
Select-Object -Property count,Name | Sort-Object -Property count |  
Format-Table -AutoSize
```

The output is a table with two columns: "Count" and "Name". The data is as follows:

Count	Name
1	Tcpip
1	Microsoft-Windows-DirectoryServices-DSROLE-Server
1	Schannel
1	volmgr
2	Microsoft-Windows-Diagnostics-Networking
2	Microsoft-Windows-Setup
2	Microsoft-Windows-GroupPolicy
2	Kerberos
2	HTTP
3	NETLOGON

Figure 1-6

Every event log includes a source that indicates where the event originated. All we've done is look at the System event log, grouped the event records by the **Source** property, and piped that result to **Select-Object** so that we only receive the **Count** and **Name** properties, which in turn are sorted by the

**Sort-Object** cmdlet, and finally PowerShell presents the results by using the **Format-Table** cmdlet. This is a terrific example of leveraging the pipeline. You can use the same technique to retrieve a summary breakdown of error types.

```
Get-EventLog -Log system | group EntryType | select Count, Name | sort count -Descending | Format-Table -AutoSize
```

If you want results for all logs, it takes a little more finesse.

```
foreach ($log in (Get-EventLog -List)) {
    #only display logs with records
    if ($log.Entries.Count -gt 0) {
        Write-Host -BackgroundColor DarkGreen -ForegroundColor Black $log.log

        Get-EventLog -LogName $log.log |
            Group-Object -Property EntryType |
            Select-Object -Property Count, Name |
            Sort-Object -Property count -Descending |
            Format-Table -AutoSize
    }
}
```

This snippet uses the **ForEach** script command to retrieve every event log on the local system, if the number of entries in each log is greater than 0.

```
if ($log.Entries.Count -gt 0) {
```

We'll display the log name, and then use the same code from earlier to return a count of each error type.

Let's combine both of our efforts and retrieve a listing of event types for each source from every event log with records.

```
foreach ($log in (Get-EventLog -List)) {
    #only display logs with records
    if ($log.Entries.Count -gt 0) {
        Write-Host -BackgroundColor DarkGreen -ForegroundColor Black $log.log

        Get-EventLog -LogName $log.log |
            Group-Object -Property source, entrytype |
            Select-Object -Property Count, Name |
            Sort-Object -Property count -Descending |
            Format-Table -AutoSize
    }
}
```

We'll receive a listing like the one shown in Figure 1-7 for every event log.

```

Administrator: Windows PowerShell
>>
Active Directory Web Services
Count Name
-----
10 ADWS, Error
11 ADWS, Warning
81 ADWS, Information

Application
Count Name
-----
1 Software Protection Platform Service, Error
1 AutoEnrollment, Error

```

Figure 1-7

## Windows 7 and Later

The **Get-Eventlog** cmdlet is designed to work with the classic logs like System, Application, and Security. This is actually true for any **-Eventlog** cmdlet. To manage the newer event logs introduced in Windows 7 and Server 2008, you need to use the **Get-WinEvent** cmdlet. This cmdlet also requires that you have at least .NET Framework version 3.5 installed. You can manage classic event logs with this cmdlet, so you may prefer to use it for all your event-log management.

Let's begin by finding out what logs are available.

```
Get-WinEvent -ListLog *
```

The **-ListLog** parameter requires a value or a wildcard. If you run this on a computer running Vista or later, you might be surprised by the number of logs. Classic logs are usually listed first. Let's look at a specific log in more detail.

```
$log=Get-Winevent -ListLog "Windows PowerShell"
Get-WinEvent -ListLog $log.logname | select *
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$log=Get-WinEvent -ListLog "Windows PowerShell" is run, followed by PS C:\> Get-WinEvent -ListLog \$log.logname | Select \*. The output displays properties of the log file, including FileSize, IsLogFull, LastAccessTime, LastWriteTime, OldestRecordNumber, RecordCount, LogName, LogType, LogIsolation, IsEnabled, IsClassicLog, and SecurityDescriptor.

Property	Value
FileSize	: 1118208
IsLogFull	: False
LastAccessTime	: 6/26/2013 5:00:13 PM
LastWriteTime	: 7/2/2013 8:41:10 AM
OldestRecordNumber	: 1
RecordCount	: 880
LogName	: Windows PowerShell
LogType	: Administrative
LogIsolation	: Application
.IsEnabled	: True
IsClassicLog	: True
SecurityDescriptor	: O:BAG:SYD:(A;;0x2;;;S-1-15-2-1)(A;;0

Figure 1-8

Now, we can easily retrieve events.

```
Get-WinEvent -LogName $log.logname
```

This code snippet returns a number of entries. Fortunately, the **Get-WinEvent** cmdlet has some parameters available to filter results. We can use the **-MaxEvents** parameter to return only 10 events.

```
Get-WinEvent -LogName $log.logname -MaxEvents 10
```

By default, the cmdlet returns newest events first. However, you can use the **-Oldest** parameter to reverse that.

```
Get-WinEvent -LogName $log.logname -MaxEvents 10 -Oldest
```

The **Get-WinEvent** cmdlet offers several filtering techniques. You could pipe output to the **Where-Object** cmdlet.

```
Get-WinEvent -LogName $log.logname |
  Where { $_.id -eq 400 -and $_.TimeCreated -gt [DateTime]"4/10/2009" -and
          $_.timeCreated -lt [DateTime]"4/11/2009" }
```

However, a better approach is to use the **-FilterHashTable** parameter and supply a hash table of properties to filter.

---

**Note:**

This is for systems running Windows 7, Windows Server 2008 R2, or later.

```
Get-WinEvent -FilterHashtable @{ LogName = $log.logname; id = 400; StartTime = "4/10/2009" }
```

When using a hash table, you must provide a **logname**, **providername**, or **path**. You are limited to the following parameters:

- LogName
- ProviderName
- Path
- Keywords
- ID
- Level
- StartTime
- EndTime
- UserID
- Data

You can create XML filters, although they are a little more complicated.

```
Get-WinEvent -FilterXML "<QueryList><Query><Select Path='Windows PowerShell'>*[System[(EventID=400)]]</Select></Query></QueryList>"
```

---

---

**Note:**

When using -FilterXML or -FilterXPath, the query is case sensitive. Change one of the tags, such as **<Query>** to **<query>**, and it will generate an error.

We recommend you use the EventViewer management console to create a query and then copy and paste the XML into your PowerShell code.

A variety of providers write Windows event logs and often you need to filter based on a given provider. First, use the **-ListProvider** parameter to see all providers.

```
Get-WinEvent -ListProvider *
```

If you noticed earlier, providers are also part of the log file's properties. There is an easy way to list them.

```
(Get-WinEvent -ListLog System).ProviderNames
```

To learn more about a specific provider, all we have to do is ask.

```
Get-WinEvent -ListProvider *firewall | Select *
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Get-WinEvent -ListProvider \*firewall | select \* is run. The output displays the properties of the Microsoft-Windows-Firewall provider:

PropertyName	:	Microsoft-Windows-Firewall
Name	:	Microsoft-Windows-Firewall
Id	:	e595f735-b42a-494b-afcd-b68666945cd3
MessageFilePath	:	C:\Windows\system32\mpssvc.dll
ResourceFilePath	:	C:\Windows\System32\mpssvc.dll
ParameterFilePath	:	
HelpLink	:	<a href="http://go.microsoft.com/fwlink/events.asp?CoName=Microsoft Corporation&amp;ProdName=Microsoft® Windows® Operating System&amp;ProdVer=6.3.9431.0&amp;FileName=mpssvc.dll&amp;FileVer=6.3.9431.0">http://go.microsoft.com/fwlink/events.asp?CoName=Microsoft Corporation&amp;ProdName=Microsoft® Windows® Operating System&amp;ProdVer=6.3.9431.0&amp;FileName=mpssvc.dll&amp;FileVer=6.3.9431.0</a>
DisplayName	:	Microsoft-Windows-Firewall
LogLinks	:	{System}
Levels	:	{win:Error}

Figure 1-9

Finally, to find events associated with a specific provider, use the **-ProviderName** parameter.

```
Get-WinEvent -ProviderName "user32"
```

## Working with remote event logs

If you have computers running Windows 7 or later that are also running .NET Framework version 3.5 or later, you can use the **Get-WinEvent** cmdlet to manage their event logs, just as if they were local. You can use the **-ComputerName** and **-Credential** parameters.

```
Get-WinEvent -ListLog * -Computer win2k801 -Credential $cred |
Where { $_.RecordCount -gt 0 } |
Sort RecordCount -Descending |
Select Logname,RecordCount, LastWriteTime, IsLogFull |
Format-Table -AutoSize
```

The **-Credential** parameter accepts a standard **PSCredential**. The **\$cred** variable is a saved **PSCredential** object.

You can use just about all the expressions we showed you earlier with remote computers; for example, to use a hash table filter, do the following:

```
Get-WinEvent -FilterHashtable @{ LogName="System"; Level = 2 } -Computer Win2k801 -Max 10
```

You can still remotely manage logs on legacy systems by using the **Get-Eventlog** cmdlet. It too has a **-ComputerName** parameter (but unfortunately, no **-Credential** parameter).

```
Get-EventLog -List -ComputerName "xp01"
```

Here's another example.

```
Get-EventLog -ComputerName "XP01" -LogName "System" -EntryType "Error" -Newest 50
```

Index	Time	EntryType	Source	InstanceID
1450	Jun 29 11:04	Error	NetJoin	4097
1199	Jun 29 10:57	Error	NETLOGON	5788
1198	Jun 29 10:57	Error	NETLOGON	5789
1105	Jun 29 10:56	Error	NETLOGON	5788
1104	Jun 29 10:56	Error	NETLOGON	5789
1099	Jun 29 10:56	Error	NetJoin	4097
1084	Jun 29 10:54	Error	NETLOGON	5788
1083	Jun 29 10:54	Error	NETLOGON	5789
1063	Jun 29 10:49	Error	Microsoft-Windows...	1129
1057	Jun 29 10:48	Error	NETLOGON	3210
911	Jun 29 10:41	Error	Microsoft-Windows...	1129

Figure 1-10

While we think PowerShell should be all you need, the EventViewer management console that ships with Windows 7 is a considerable improvement over the legacy Microsoft Management Console (MMC). PowerShell includes the **Show-EventLog** cmdlet that will launch the EventViewer.

**Show-EventLog**

You can also use the **-ComputerName** parameter to connect to a remote computer.

## Configuring event logs

PowerShell version 4 offers a few cmdlets for configuring event logs themselves. For example, to change the Windows PowerShell log to have a size limit of 10 MB, do the following:

```
Limit-EventLog -LogName "Windows PowerShell" -MaximumSize 10MB
```

Perhaps you also want to change the log so that it overwrites as needed and retains records for no longer than 14 days.

```
Limit-EventLog -LogName "Windows PowerShell" -OverflowAction "overwriteolder" -RetentionDays 14
```

Configure logs on remote servers by using the **-ComputerName** parameter.

```
Get-Content servers.txt | foreach { Limit-Eventlog -Logname System -Computername $_ -MaximumSize 25MB }
```

## Backup event logs

Unfortunately, there are still no cmdlets for backing up classic event logs in PowerShell version 2. However, backing up event logs is relatively straightforward in PowerShell. If you use the [WMI] type adapter, you'll have access to the **BackupEventLog()** method.

```
[wmi]$syslog = Get-WMiobject -Query "Select * from win32_NTEventLogFile where ` 
LogFileNames='system'" -EnableAllPrivileges

$backup=(Get-Date -Format yyyyMMdd) + "_" + $syslog.CSName + "_" + $syslog.logfilename + ".evt"

$syslog.backupeventlog("F:\backups\$backup")
```

Here's how this works. In this example, we are going to back up the System event log on the local computer.

```
[wmi]$syslog = Get-WMiobject -Query "Select * from win32_NTEventLogFile where ` 
LogFileNames='system'"
```

We need a name for the backup file, which we calculate using the current date, the name of the computer, and the log file name.

```
$backup = (Get-Date -Format yyyyMMdd) + "_" + $syslog.CSName + "_" + $syslog.logfilename + ".evt"
```

This expression will return a value like 20090422\_XPDESK01\_System.evt. The advantage to using the **CSName** property is that if we back up a remote server, we can automatically capture the name.

In order to back up event logs, you need to specify the **Backup** privilege. If you don't, you'll receive an Access Denied message when you try to back up the log. That's why we include the **-EnableAllPrivileges** parameter.

With privileges enabled, we can now back up the event log.

```
$syslog.backupeventlog("F:\backups\$backup")
```

## Location, location, location

There is a very subtle but important detail about the **BackupEventLog()** method, especially when using the **Get-WmiObject** cmdlet to access event logs on remote computers. Even though you are running a script and remotely accessing a computer, the backup method is actually executing on the remote system. This means that the path you specify is relative to the remote system. If you back up the event log to drive C:\, it will be backed up to drive C:\ of the remote computer, not the computer where you are executing the script. Verify that the destination folder is accessible from the remote computer and you won't have any surprises.

If you want to back up all event logs, you can use code like the following:

```
$path="F:\Backups"
foreach ($log in (Get-WmiObject win32_nteventlogfile -Enableallprivileges)) {
    $backup = (Get-Date -Format yyyyMMdd) + "_" + $log.CSName + "_" + $log.logfilename + ".evt"
    Write-Host "Backing up"$log.LogFileName"to $path\$backup"
    $rc = $log.backupeventlog($path + "\" + $backup)
    if ($rc.ReturnValue -eq 0) {
        Write-Host "Backup successful" -Foreground GREEN }
        else {
            Write-Host "Backup failed with a return value of"$rc.ReturnValue -Foreground RED
    }
}
```

The **\$path** variable is the backup directory that we want to use. Using a **Foreach** loop, we can return every event log on the computer, enabling all privileges.

```
foreach ($log in (Get-WmiObject win32_nteventlogfile -RnableAllPrivileges)) {
```

As we did before, we define a backup file name.

```
$backup = (Get-Date -Format yyyyMMdd) + "_" + $log.CSName + "_" + $log.logfilename + ".evt"
Write-Host "Backing up"$log.LogFileName"to $path\$backup"
```

When we call the **BackupEventLog()** method, we save the results to a variable.

```
$rc = $log.backupeventlog($path + "\\" + $backup)
```

With this variable, we can check if the backup was successful or not. If it wasn't, we display the **ReturnValue** property.

```
if ($rc.ReturnValue -eq 0) {
    Write-Host "Backup successful" -Foreground GREEN }
else {
    Write-Host "Backup failed with a return value of"$rc.ReturnValue -Foreground RED
}
```

One thing to be aware of is that if the backup file already exists, it will not be overwritten and you will receive a return value of 183.

## Clearing event logs

Clearing an event log is very easy in PowerShell version 4, by using the **Clear-EventLog** cmdlet.

```
Clear-EventLog -LogName "Windows Powershell"
```

There is a slight caveat: this only works with classic logs. However, you can use the **-ComputerName** parameter and clear logs on remote computers. We're not aware of any PowerShell solutions for backing up or clearing the newer event-log types that shipped with Windows 7 and later.

Let's wrap up our exploration of backing up and clearing event logs, by using a PowerShell script. The **Backup-EventLog.ps1** script uses WMI to back up and clear event logs on any computer in your domain.

### **Backup-EventLog.ps1**

```
<#
.Synopsis
    Back up a computer event log with an option to clear.
.Description
    This function will back up the specified event log on a given computer
    to the specified path. Each log will be backed up to a file named
    YYYYMMDD_computername_logname.evt. You can only back up event logs
    to a local drive, relative to the remote computer. You'll need a
    separate process to copy or move the file to a network share. If
    you are running this on the local machine, then you can specify a mapped
    drive or UNC.

    If you specify -clear, each backed up log will also be cleared.
.Parameter Computername
    What is the name of the computer to back up? The default is the local computer.
```

```

.Parameter Logname
    What is the name of the event log to back up? The backup will fail if the
    file already exists.
.Parameter Filepath
    What is the path for the backup file? This path is relative to the remote
    computer and must be a local drive.
.Parameter Clear
    Clear the event log if it successfully backed up
.Example
    Backup-Eventlog -computername CHAOS -logname "Windows PowerShell" -Filepath "d:\backups"
    -clear

This backs up the Windows PowerShell log to the D:\Backups drive on CHAOS.
The log will be cleared if the backup is successful.
.Example
    PS C:\> Get-Eventlog -List -AsString | foreach { .\Backup-EventLog.ps1 -Logname $_ -Filepath
    q:\backup}

    This gets all event logs on the local computer and backs them up to Q:\backup
.Example
    PS C:\> Get-Content servers.txt | foreach {.\backup-eventlog.ps1 -Computer $_ -Log Security
    -Filepath c:\ -Clear}

    This command will process the list of server names. Each name is piped to Foreach which
connects
    to the remote computer, backs up the Security event log to the local C: drive and then clears
the log
    if successful.

.ReturnValue
    None
.Link
    Get-WMIObject
    Get-Eventlog
    Clear-Eventlog

.Notes
    NAME:      Backup-EventLog
    VERSION:   1.0
    AUTHOR:    Jeffery Hicks
    LASTEDIT:  4/16/2009 5:00:00 PM

#requires -version 2.0
#>

[CmdletBinding(
    SupportsShouldProcess = $False,
    SupportsTransactions = $False,
    ConfirmImpact = "None",
    DefaultParameterSetName = "")]

param(
[Parameter(Position = 0, Mandatory = $False, ValueFromPipeline = $false,
    HelpMessage="What is the name of the computer to back up?")]
[string]$computername = $env:computername,

[Parameter(Position = 1, Mandatory = $true, ValueFromPipeline = $false,
    HelpMessage="Enter an event log name like System")]
[string]$logname,

[Parameter(Position = 2, Mandatory=$true, ValueFromPipeline = $false,
    HelpMessage="Enter the path for the backup log.")]
[string]$filepath,

```

```
[Parameter(Position = 3, Mandatory = $false, ValueFromPipeline = $false)]
[switch]$Clear
)

#main script body
$log = Get-WmiObject -Class win32_NTEventLogFile -Computername $computername -Authentication
PacketPrivacy -EnableAllPrivileges -Filter "logfilename='$logname'"

if ($log) {
    $backupfile = "{0}_{1}_{2}.{3}" -f (Get-Date -Format yyyyMMdd), $computername,
    $logname.replace(" ", ""), $log.extension
    $backup = Join-Path $filepath $backupfile

    Write-Host "Backing up $($log.name) on $computername to $backup" -Foreground Cyan

    $rc = $log.BackupEventLog($backup)

    switch ($rc.ReturnValue) {
        0 { #success
            Write-Host "Backup successful." -Foreground Green
            if ($clear) {
                $rc = $log.ClearEventLog()
                if ($rc.returnValue -eq 0) {
                    Write-Host "Event log cleared" -Foreground Green
                }
                else {
                    Write-Warning "Failed to clear event log.Return code $($rc.ReturnValue)" ` -Foreground Green
                }
            } #end if $clear
        } #end 0
        3 {
            Write-Warning "Backup failed. Verify $filepath is accessible from $computername"
        }
        5 {
            Write-Warning "Backup failed. There is likely a permission or delegation problem."
        }
        8 {
            Write-Warning "Backup failed. You are likely missing a privilege."
        }
        183 {
            Write-Warning "Backup failed. A file with the same name already exists."
        }
        Default {
            Write-Warning "Unknown Error! Return code $($rc.ReturnValue)"
        }
    }#end switch
}

else {
    Write-Warning "Failed to find $logname on $computername"
}
```

The script takes advantage of scripting features introduced in PowerShell version 2. The first section defines script metadata that you can use with the Help feature.

```
help .\backup-eventlog.ps1
```

As you can see, the script uses a computer and log file name as parameters. You must also specify a loca-

tion for the backed up event log. This directory is relative to the remote computer and must be a local drive. If you are running the script on your computer, you can specify a networked drive or UNC.

The script uses the **Get-WmiObject** cmdlet to retrieve the event log.

```
$log = Get-WmiObject -Class win32_NTEventLogFile -Computername $computername` -Authentication
PacketPrivacy -EnableAllPrivileges -Filter "logfilename='\$logname'"
```

The script constructs the backup file name if it finds a log file.

```
if ($log) {
    $backupfile = "{0}_{1}_{2}.{3}" -f (Get-Date -Format yyyyMMdd), $computername,`  

    $logname.replace(" ",""), $log.extension
    $backup = Join-Path $filepath $backupfile
```

The script removes spaces from the log file name. The backup file name will be something like 20090418\_XP01\_WindowsPowerShell.evt. The backup file will use the same file extension as the log file. You join this file name to the file path by using the **Join-Path** cmdlet, to create a complete file name.

PowerShell then uses the file name as a parameter for the **BackupEventLog()** method.

```
$rc = $log.BackupEventLog($backup)
```

This method returns an object and we can use its **ReturnValue** to determine whether the backup was successful. A **Switch** construct will display different warning messages, depending on the return value.

Assuming a successful backup, if you used the **-Clear** parameter, then PowerShell clears the log, by using the **ClearEventLog()** method. Again, we capture the return object so we can tell whether this was successful.

```
switch ($rc.ReturnValue) {
    0 { #success
        Write-Host "Backup successful." -Foreground Green
        if ($clear) {
            $rc = $log.ClearEventLog()
            if ($rc.returnValue -eq 0) {
                Write-Host "Event log cleared" -Foreground Green
            }
            else {
                Write-Warning "Failed to clear event log.Return code $($rc.ReturnValue)" `  

                -Foreground Green
            }
        } #end if $clear
    } #end 0
```

Here are some ways you might use this script.

```
Backup-Eventlog -Computername CHAOS -Logname "Windows PowerShell" -Filepath "d:\backups" -Clear
```

This expression will back up the Windows PowerShell log on CHAOS to the D:\Backups folder. PowerShell will clear the log if the backup is successful.

```
Get-EventLog -List -AsString | foreach { .\Backup-EventLog.ps1 -Logname $_ -Filepath q:\backup }
```

In the previous example, we are retrieving all event logs on the local computer, and then backing them up to Q:\backup.

```
Get-Content servers.txt | foreach {.\backup-eventlog.ps1 -Computer $_ -Logname Security ` -Filepath c:\ -clear }
```

This command will process the list of server names in servers.txt. We pipe each name to the **Foreach** loop, which connects to the remote computer, backs up the Security event log to the remote computer's C:\ drive, and then clears the log if successful.

## In Depth 02

# Managing files and folders

## Managing files and folders

File and directory management is a pretty common administrative task. Here are some ways to accomplish typical file and directory management tasks by using PowerShell.

## Creating text files

In the beginning Learning Guide section of the book, you discovered how to use console redirection to send output to a text file and how to use the **Out-File** cmdlet, so we won't go into detail about that. Instead, we'll provide a few examples for creating text files. First, let's look at a line of code that redirects output to a text file.

```
Get-WmiObject -Class win32_share > c:\MyShares.txt
```

```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject -Class win32_share > c:\MyShares.txt
PS C:\> Get-Content c:\myshares.txt

Name          Path          Description
----          ---          -----
ADMIN$        C:\Windows    Remote Admin
C$           C:\           Default share
IPC$          C:\Windows\SYSVOL\... Logon server share
NETLOGON      C:\Windows\SYSVOL\... Logon server share
SYSVOL

PS C:\>
```

Name	Path	Description
ADMIN\$	C:\Windows	Remote Admin
C\$	C:\	Default share
IPC\$	C:\Windows\SYSVOL\...	Logon server share
NETLOGON	C:\Windows\SYSVOL\...	Logon server share
SYSVOL		

Figure 2-1

Remember that `>` sends output to the specified file, overwriting any existing file with the same name.

Here's the same expression, but using the **Out-File** cmdlet.

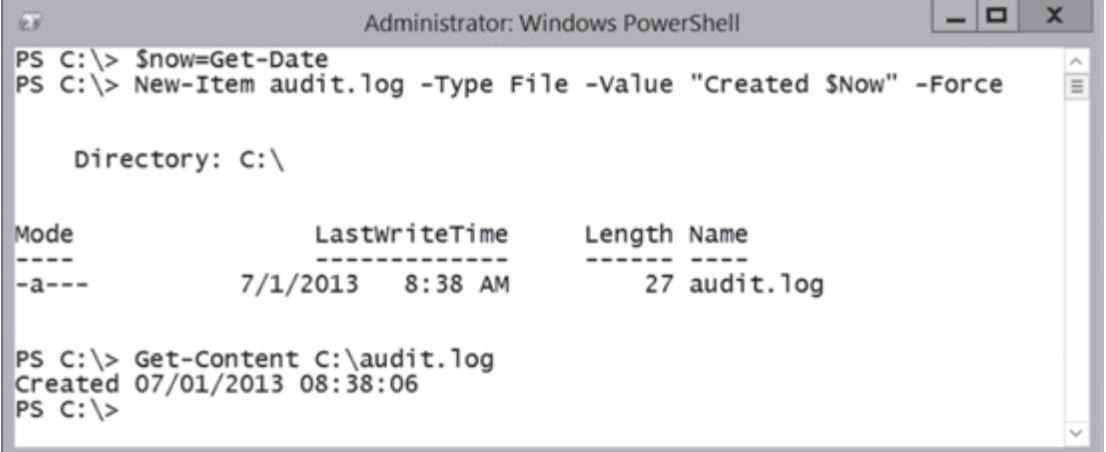
```
Get-WmiObject -Class win32_share | Out-File -PSPPath c:\MyShares.txt -NoClobber
```

Use `>>` or the **-Append** parameter of **Out-File** to append output to an existing file. The file will be created if it doesn't already exist.

We used the **-NoClobber** parameter to prevent the **Out-File** cmdlet from overwriting `myshares.txt`, if it already exists.

Finally, you can also use the **New-Item** cmdlet to create a file and even add some content to it.

```
$now = Get-Date
New-Item audit.log -Type File -Value "Created $Now" -Force
```



The screenshot shows an Administrator: Windows PowerShell window. The command PS C:\> \$Now=Get-Date is entered. Then, PS C:\> New-Item audit.log -Type File -Value "Created \$Now" -Force is run. A table lists the file 'audit.log' with Mode -a---, LastWriteTime 7/1/2013 8:38 AM, Length 27, and Name audit.log. Finally, PS C:\> Get-Content C:\audit.log shows the content 'Created 07/01/2013 08:38:06'.

Figure 2-2

In Figure 2-2, we used the **New-Item** cmdlet to create a file called audit.log and to give it some content.

## Reading text files

Reading text files is pretty straightforward with the **Get-Content** cmdlet.

```
get-content c:\MyShares.txt
```

We can use the **-TotalCount** parameter to display a specified number of lines from a text file.

```
get-content c:\MyShares.txt -TotalCount 4
```

```

Administrator: Windows PowerShell
PS C:\> get-content c:\MyShares.txt
Name          Path          Description
----          ---          -----
ADMIN$        C:\Windows    Remote Admin
C$           C:\           Default share
IPC$          C:\Windows\SYSVOL\...  Remote IPC
NETLOGON      C:\Windows\SYSVOL\...  Logon server share
SYSVOL       C:\Windows\SYSVOL\...  Logon server share

PS C:\> get-content c:\MyShares.txt -TotalCount 4
Name          Path          Description
----          ---          -----
ADMIN$        C:\Windows    Remote Admin
PS C:\>

```

Figure 2-3

In Figure 2-3, the first four lines of a log file are displayed. You can receive the same result with the expression in Figure 2-4.

```
get-content c:\MyShares.txt | Select -First 4
```

```

Administrator: Windows PowerShell
PS C:\> get-content c:\MyShares.txt | Select -First 4
Name          Path          Description
----          ---          -----
ADMIN$        C:\Windows    Remote Admin
PS C:\>

```

Figure 2-4

An advantage to this approach is that the **Select-Object** cmdlet also has a **-Last** parameter, which you can use to display a specified number of lines from the end of the file. Be careful with this—if your file has a million rows, the entire contents will be read just to show the last four lines.

## Parsing text files

While PowerShell might be a highly object-oriented shell, that doesn't mean you don't need to work with pure text now and again. Log files and INI files are perhaps two of the most common examples of text files Windows admins need to work with on a regular basis.

Typically, in a text file, you are looking to extract pieces of information and PowerShell offers several techniques for this. The simplest approach is to use the **Match** operator.

```
(Get-Content c:\MyShares.txt) -match "windows"
```

Another way is to pipe your files to the **Select-String** cmdlet.

```
Get-Content C:\MyShares.txt | Select-String -SimpleMatch "windows"
```

```

Administrator: Windows PowerShell
PS C:\> (Get-Content c:\MyShares.txt) -match "windows"
ADMIN$           C:\Windows          Remote Admin
NETLOGON         C:\Windows\SYSVOL\s... Logon server share
SYSVOL          C:\Windows\SYSVOL\s... Logon server share
PS C:\> Get-Content C:\MyShares.txt | Select-String -Pattern "windows"

ADMIN$           C:\Windows          Remote Admin
NETLOGON         C:\Windows\SYSVOL\s... Logon server share
SYSVOL          C:\Windows\SYSVOL\s... Logon server share

PS C:\> -

```

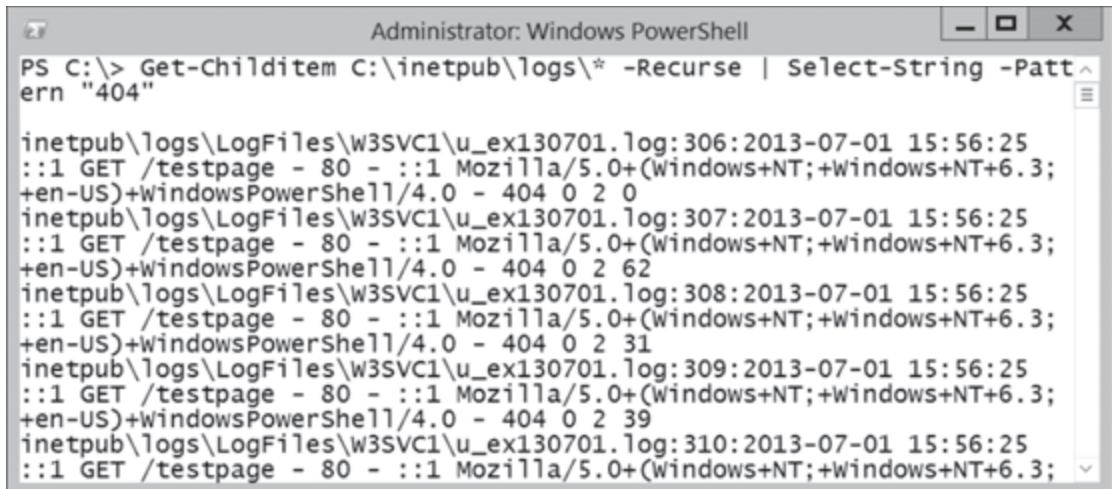
Figure 2-5

More complicated parsing and matching might require regular expressions and the **-Pattern** parameter.

## Parsing IIS log files

Using **Select-String**, it is very easy to extract information from IIS log files. Suppose you want to find all 404 errors that occurred during July 2013. You could use an expression like the one in Figure 2-6.

```
Get-ChildItem C:\inetpub\logs\* -Recurse | Select-String -Pattern "404"
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Get-Childitem C:\inetpub\logs\* -Recurse | Select-String -Pattern "404"
```

The output lists several log entries from the IIS logs, all containing the string "404".

```
inetpub\logs\LogFiles\W3SVC1\u_ex130701.log:306:2013-07-01 15:56:25
::1 GET /testpage - 80 - ::1 Mozilla/5.0+(Windows+NT;+Windows+NT+6.3;
+en-US)+WindowsPowerShell/4.0 - 404 0 2 0
inetpub\logs\LogFiles\W3SVC1\u_ex130701.log:307:2013-07-01 15:56:25
::1 GET /testpage - 80 - ::1 Mozilla/5.0+(Windows+NT;+Windows+NT+6.3;
+en-US)+WindowsPowerShell/4.0 - 404 0 2 62
inetpub\logs\LogFiles\W3SVC1\u_ex130701.log:308:2013-07-01 15:56:25
::1 GET /testpage - 80 - ::1 Mozilla/5.0+(Windows+NT;+Windows+NT+6.3;
+en-US)+WindowsPowerShell/4.0 - 404 0 2 31
inetpub\logs\LogFiles\W3SVC1\u_ex130701.log:309:2013-07-01 15:56:25
::1 GET /testpage - 80 - ::1 Mozilla/5.0+(Windows+NT;+Windows+NT+6.3;
+en-US)+WindowsPowerShell/4.0 - 404 0 2 39
inetpub\logs\LogFiles\W3SVC1\u_ex130701.log:310:2013-07-01 15:56:25
::1 GET /testpage - 80 - ::1 Mozilla/5.0+(Windows+NT;+Windows+NT+6.3;
```

Figure 2-6

You'll see a listing of every matching line. Use console redirection or the **Out-File** cmdlet to save the results.

```
Get-Childitem C:\inetpub\logs\* -Recurse | Select-String -Pattern "404" | Out-File C:\404errors.txt
```

Let's take this a step further and find the IP addresses for each computer that received a 404 error. To accomplish this, we will use a combination of the **Select-String** cmdlet and a regular expression.

```
[regex]$regex = "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"

$(foreach ($found in (Get-Childitem ex0705* |
select-string " - 404")) {
($regex.matches($found.ToString()))[1].value} ) | select -Unique
```

We'll break this apart from the inside out so you can see what is happening.

```
Get-Childitem ex0705* | Select-String " - 404"
```

We know that an expression like the previous example will return all the strings from log files that start with "ex0705" that match on "- 404". We want to examine each matched line, so we'll nest the previous command in a **Foreach** construct.

```
Foreach($found in (Get-Childitem ex0705* | Select-String " - 404"))
```

For every line in the file, we first need to convert it into a string.

```
$found.ToString()
```

We have a **regex** object, with a pattern that should match an IP address.

```
[regex]$regex = "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"
```

The **Matches()** method of the **Regex** object will return all matching instances in the line.

```
$regex.matches($found.ToString())
```

However, we only need to see the value of the second match. The first match will be the web server's IP address.

```
($regex.matches($found.ToString()))[1].value
```

Now, let's run what we have so far.

```
foreach ($found in (Get-Childitem ex0705* | `  
Select-String " - 404")) { ($regex.matches($found.ToString()))[1].value }
```

With the previous example, every client IP address would be listed, probably in duplicate. Because this is PowerShell, we'll add one final tweak by piping the output of this command to the **Select-Object** cmdlet and specifying the **-Unique** parameter.

```
$(foreach ($found in (Get-Childitem ex0705* | `  
Select-String " - 404")) {  
($regex.matches($found.ToString()))[1].value } ) | select -Unique
```

Notice that we've enclosed the main filtering expression in parentheses and preceded it by a \$ sign. This indicates that PowerShell should treat it as an expression and execute it in its entirety. We then piped the results of that expression to the **Select-Object** cmdlet, leaving us with a list of unique IP addresses.

## Parsing INI files

Another common administrative task is parsing information from INI files, is shown in the following example.

```
;MyApp.ini
;last updated 2:28 PM 6/13/2008
[Parameters]
Password=P@ssw0rd
Secure=Yes
Encryption=Yes
UseExtendedSyntax=No

[Open_Path]
path=\server01\files\results.txt

[mail]
server=MAIL01
from=admin@mycompany.com

[Win2003]
foo=bar
```

If all the settings are unique in the entire file, you can use the **Select-String** cmdlet to extract the value for a particular setting with a PowerShell one-liner.

```
((cat myapp.ini | Select-String -Pattern "password=").ToString()).Split("=")[1]
```

Here's how this works. First, this example is using **Cat**, which is an alias for the **Get-Content** cmdlet, which keeps a long expression from having to be any longer. Piping the contents of myapp.ini to the **Select-String** cmdlet will return a **MatchInfo** object when PowerShell finds a match.

```
cat myapp.ini | Select-String -Pattern "password="
```

However, we need to convert the result to a string object, so we use the **Tostring()** method.

```
(cat myapp.ini | Select-String -Pattern "password=").ToString()
```

If we were to look at the returned value thus far, we would see the following:

```
Password=P@ssw0rd
```

Since we now have a string, we can use the **Split()** method to split the string at the = sign.

```
((cat myapp.ini | Select-String -Pattern "password=").ToString()).Split("=")
```

This expression results in a small 0-based array, meaning the first element has an index number of 0. All that remains is to display the second array element, which has an index number of 1.

```
((cat myapp.ini | Select-String -Pattern "password=").ToString()).Split("=")[1]
```

In our sample INI file, this technique would work since all of our values are unique. But what about situations where we need a specific value under a specific heading? That is a little more complicated, but you can achieve it with some extra parsing. The following is a function we wrote to retrieve the value from a particular section of an INI file.

```
Function Get-INIValue {
    # $ini is the name of the ini file
    # $section is the name of the section head like [Mail]
    # Specify the name without the brackets
    # $prop is the property you want under the section
    # sample usage: $from=Get-inivalue myapp.ini mail from

    Param ([string]$ini, [string]$section, [string]$prop)

    #get the line number to start searching from
    $LineNum = (Get-Content myapp.ini | Select-String "\[$section\]").LineNumber
    $limit = (Get-Content myapp.ini).Length #total number of lines in the ini file

    for ($i = $LineNum; $i -le $limit; $i++) {
        $line = (Get-Content myapp.ini)[ $i ]
        if ($line -match $prop + "=") {
            $value = ($line.split("="))[1]
            return $value
            Break
        }
    }
    return "NotFound"
}
```

The function is expecting the name of the INI file, the section name, and the name of the value. Once PowerShell loads this file, you can use it.

```
$smtp = Get-Inivalue myapp.ini mail server
```

The function will return the value of the “server” setting under the **[Mail]** section of myapp.ini. The function first obtains the number of the lines in the INI file that contains the section heading.

```
$LineNum = (Get-Content myapp.ini | Select-String "\[$section\]").LineNumber
```

The function also retrieves the total number of lines in the INI file, that it will use later.

```
$limit = (Get-Content myapp.ini).Length #total number of lines in the ini file
```

Now that we know where to start searching, we can loop through each line of the INI file until we reach the end.

```
for ($i = $LineNum; $i -le $limit; $i++) {
```

Because the INI file will be treated as a 0-based array, the value of **\$LineNum** will actually return the first line after the heading.

```
$line = (Get-Content myapp.ini)[$i]
```

The function examines each line by using the **Match** operator to see if it contains the property value we are seeking.

```
if ($line -match $prop + "=") {
```

If not, the function will keep looping until it reaches the end of the file, at which point the function will return **NotFound**.

When PowerShell makes a match, it splits the line at the = sign and returns the second item (index number of 1).

```
$value = ($line.split("="))[1]
```

The function returns this value, and since there's no longer any need to keep looping through the INI file, the function stops, by using the **Break** statement.

```
return $value
Break
```

## Copying files

Copying files in PowerShell is not much different than copying files in Cmd.exe. In fact, by default PowerShell uses the alias **Copy** for the **Copy-Item** cmdlet.

```
Copy C:\scripts\*.ps1 c:\temp
Get-ChildItem c:\temp
```

The screenshot shows an Administrator Windows PowerShell window. The command PS C:\> Copy C:\scripts\\*.ps1 c:\temp is run, followed by PS C:\> Get-ChildItem c:\temp. The output shows a directory listing for C:\temp with the following details:

Mode	LastWriteTime	Length	Name
-a---	6/29/2013 1:09 PM	869	DiskInfo.ps1
-a---	5/3/2013 6:31 PM	57	GetEvent.ps1
-a---	5/3/2013 6:46 PM	91	GetProcess.Ps1
-a---	5/3/2013 6:36 PM	49	GetService.ps1
-a---	2/25/2012 2:12 PM	580	ShowControlPanelItem.ps1
-a---	6/28/2013 8:47 AM	11	test.ps1

Figure 2-7

This example copies all ps1 files in C: to C:\temp. You can also recurse and force files to be overwritten.

```
Copy-Item C:\Logs C:\Backup -Recurse -Force
```

This expression copies all files and subdirectories from C:\Logs to C:\Backup, overwriting any existing files and directories with the same name.

As you work with PowerShell, you'll discover that not every command you can execute in Cmd.exe is valid in PowerShell. For example, the following example is a valid command in Cmd.exe.

```
copy *.log *.old
```

However, if you try this in PowerShell, you'll receive a message about an invalid character. It appears the **Copy-Item** cmdlet works fine when copying between directories, but it can't handle a wildcard copy within the same directory.

Here's a workaround that you can use.

### BulkCopy.ps1

```
#BulkCopy.ps1
Set-Location "C:\Logs"

$files = Get-ChildItem | where {$_['extension -eq ".log"}

foreach ($file in $files) {
    $filename = ($file.FullName).ToString()
    $arr = @($filename.split("."))
    $newname = $arr[0] + ".old"

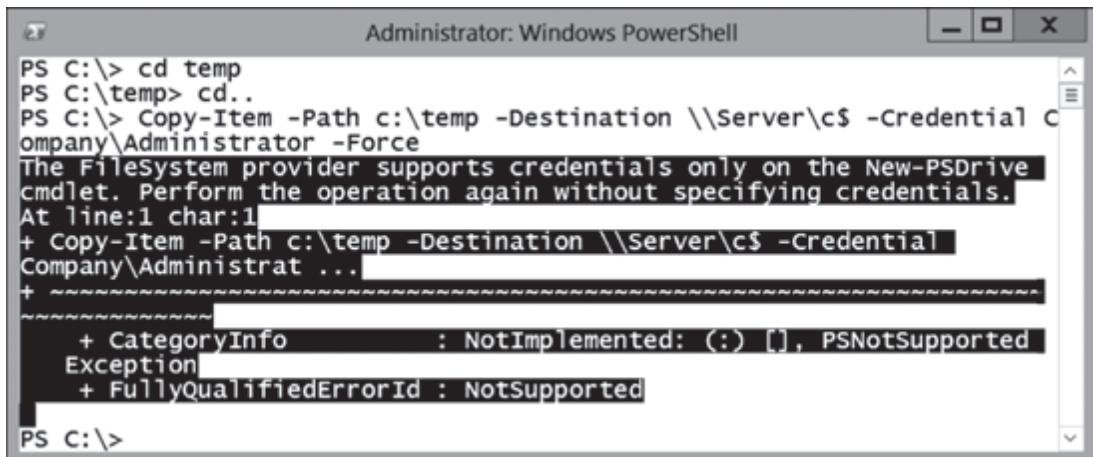
    Write-Host "copying "$file.Fullname "to"$newname
```

```
copy $file.fullname $newname -force
}
```

With this script, we first define a variable that contains all the files that we want to copy, by extension. Next, we iterate through the variable, using the **Foreach** construct. Within the loop, we break apart the filename, by using the **Split** method, so that we can get everything to the left of the period. We need this name so that we can define what the new filename will be with the new extension, including the path. Then it's a matter of calling the **Copy-Item** cmdlet. Notice that in the script, we're using the **Copy** alias.

## Provider alert

If you look through the Help for the **Copy-Item** cmdlet, and some of the other **Item** cmdlets, you will see a **-Credential** parameter. This might lead you to believe that you could use the **-Credential** parameter to copy files to a network and share and specify alternate credentials.



```
Administrator: Windows PowerShell
PS C:\> cd temp
PS C:\temp> cd..
PS C:\> Copy-Item -Path c:\temp -Destination \\Server\c$ -Credential Company\Administrator -Force
The FileSystem provider supports credentials only on the New-PSDrive cmdlet. Perform the operation again without specifying credentials.
At line:1 char:1
+ Copy-Item -Path c:\temp -Destination \\Server\c$ -Credential
Company\Administrat ...
+
+-----[REDACTED]-----
+ CategoryInfo          : NotImplemented: (:) [], PSNotSupportedException
Exception
+ FullyQualifiedErrorId : NotSupportedException
PS C:\>
```

Figure 2-8

Unfortunately, in the current version of PowerShell, the file system and registry providers do not support this parameter. Hopefully, this will change in later versions of PowerShell. In the meantime, start a PowerShell session using the **RunAs** command if you need to specify alternate credentials.

## Deleting files

The **Remove-Item** cmdlet has aliases of **del** and **erase**, and functions essentially the same as these commands in Cmd.exe.

```
Remove-Item -Path c:\temp -Filter *.ps1
```

```
Administrator: Windows PowerShell
PS C:\> Remove-Item -Path c:\temp -Filter *.ps1
Confirm
The item at C:\temp has children and the Recurse parameter was not
specified. If you continue, all children will be removed with the
item. Are you sure you want to continue?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend
[?] Help(default is "Y"): y
PS C:\> _
```

Figure 2-9

The cmdlet comes in handy when you want to recurse through a directory structure or exclude certain files.

```
Remove-Item c:\backup\*.* -Recurse -Exclude 2008*.log
```

This expression will recurse through c:\backup, deleting all files except those that match the pattern “2008\*.log”. Like the **Copy-Item** cmdlet, you can also use the **-Include** and **-Credential** parameters.

## Renaming files

Renaming files is also very straightforward.

```
Rename-Item foo.txt bar.txt
```

The **Rename-Item** cmdlet has aliases of **rni** and **ren**, and like the other item cmdlets, it lets you specify credentials and force an existing file to be overwritten. You have to be a little more creative if you need to rename multiple files.

### BulkRename.ps1

```
#BulkRename.ps1
Set-Location "C:\Logs"

$files=Get-ChildItem -Recurse |where {$_.extension -eq ".Log"}

foreach ($file in $files) {
    $filename = ($file.name).ToString()
    $arr = @($filename.split("."))
```

```
$newname = $arr[0] + ".old"
Write-Host "renaming">$file.Fullname "to">$newname
ren $file.fullname $newname -Force
}
```

The following example is a legitimate command in Cmd.exe.

```
ren *.log *.old
```

Since this doesn't work in PowerShell, we use something like the **BulkRename** script instead. This is a variation on our **BulkCopy** script from above. Instead of copy, we call **ren**. By the way, as the script is written above, it will recurse through subdirectories, starting in C:\Logs, renaming every file it finds that ends in .log to .old.

## File attributes and properties

Earlier in this chapter, we worked with file attributes to work around some issues related to copying files in PowerShell. Often times, you may need to know if a file is marked as **ReadOnly** or you will need to set it as such. You can easily accomplish this by checking the **IsReadOnly** property of the file object.

```
(Get-ChildItem file.txt).Isreadonly
False
```

You can enable **ReadOnly** by calling the **Set-ReadOnly()** method.

```
(Get-ChildItem file.txt).Set_Isreadonly($TRUE)
(Get-ChildItem file.txt).Isreadonly
True
```

Specify **\$TRUE** to enable it and **\$FALSE** to turn it off.

```
(Get-ChildItem file.txt).Set_Isreadonly($False)
(Get-ChildItem file.txt).Isreadonly
False
```

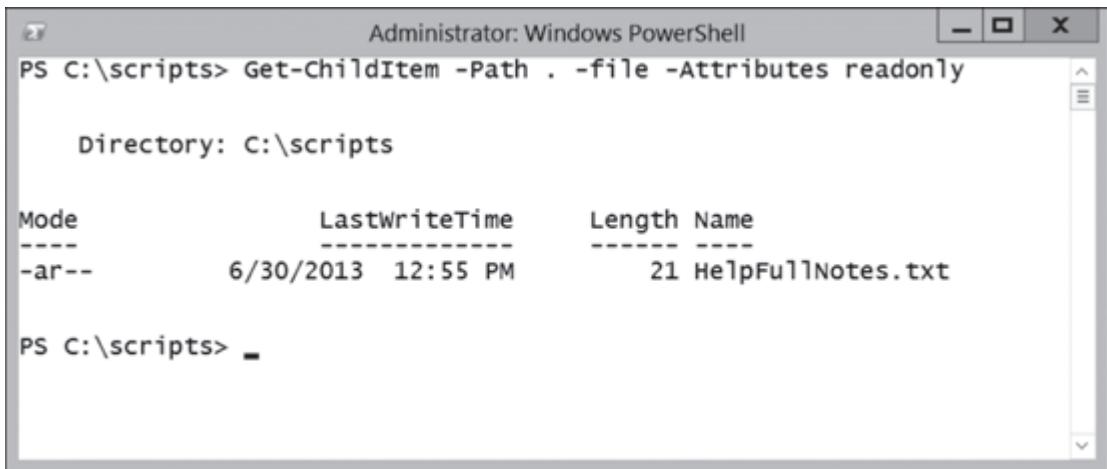
To display other attributes, you can use the **Attributes** property.

```
(Get-Childitem boot.ini -force).Attributes
Hidden, System, Archive
```

We use the **-Force** parameter so that the **Get-ChildItem** cmdlet will ignore the **Hidden** attribute and display file information.

Another method of looking for files with specific attributes was added in PowerShell version 3, with the **-Attribute** parameter. Combined with the **-File** or **-Directory** parameter, you can quickly look for files or directories that have specific attributes. To find files marked as **ReadOnly**, do the following:

```
Get-ChildItem -Path . -file -Attributes readonly
```



The screenshot shows an Administrator Windows PowerShell window. The command entered is `Get-ChildItem -Path . -file -Attributes readonly`. The output lists a single file named `HelpFullNotes.txt` located in the `C:\scripts` directory. The file has a mode of `-ar--`, a last write time of `6/30/2013 12:55 PM`, and a length of `21`.

Mode	LastWriteTime	Length	Name
<code>-ar--</code>	<code>6/30/2013 12:55 PM</code>	<code>21</code>	<code>HelpFullNotes.txt</code>

Figure 2-10

Check out the Help file for **Get-ChildItem** for more details on attributes. You can use logical operators to alter your queries. For example, if you want to list all files in a directory that are not read-only, do the following:

```
Get-ChildItem -Path . -file -Attributes !readonly
```

```
Administrator: Windows PowerShell
PS C:\scripts> Get-ChildItem -Path . -file -Attributes !readonly

Directory: C:\scripts

Mode                LastWriteTime     Length Name
----              -----     ----- 
-a---       6/29/2013 1:09 PM      869 DiskInfo.ps1
-a---       5/3/2013 6:31 PM       57 GetEvent.ps1
-a---       5/3/2013 6:46 PM       91 GetProcess.Ps1
-a---       5/3/2013 6:36 PM       49 GetService.ps1
-a---       7/1/2013 9:38 AM      197 scripts.recall
-a---      2/25/2012 2:12 PM      580 ShowControlPanelItem.ps1
-a---       6/28/2013 8:47 AM       11 test.ps1
```

Figure 2-11

## Setting attributes

To set file attributes, you can use the **Set\_Attributes()** method.

```
(Get-ChildItem file.txt).Attributes
Archive

(Get-ChildItem file.txt).Set_Attributes("Archive,Hidden")
(Get-ChildItem file.txt -force).Attributes
Hidden, Archive
```

In this snippet, you can see that file.txt only has the **Archive** attribute set. Using **Set\_Attributes()**, we set the file attributes to **Archive** and **Hidden**, which is confirmed with the last expression. This is another way of setting the **ReadOnly** attribute.

Setting the file attributes to **Normal** will clear all basic file attributes.

```
(Get-ChildItem file.txt -Force).Set_Attributes("Normal")
(Get-ChildItem file.txt -Force).Attributes
Normal
```

Since everything is an object in PowerShell, working with file properties, such as when a file was created or last modified, is very simple. Here's an abbreviated output that retrieves all files from C:\Temp and displays when the file was created, last accessed, and last modified.

```
Get-ChildItem c:\temp | Select -Property Name, CreationTime, LastAccessTime, LastWriteTime
```

The screenshot shows an Administrator: Windows PowerShell window. The command run is `Get-ChildItem c:\temp | Select -Property Name,CreationTime,LastAccessTime,LastWriteTime`. The output is a table:

Name	CreationTime	LastAccessTime	LastWriteTime
1015_SSV_Confi...	7/1/2013 10:06...	7/1/2013 10:0...	9/28/2012 9:5...
ConfigVMSwitch...	7/1/2013 10:06...	7/1/2013 10:0...	9/18/2012 10:...
IIS8Certificat...	7/1/2013 10:06...	7/1/2013 10:0...	1/3/2013 12:5...

Figure 2-12

## If isn't necessarily what you think

The **LastAccessTime** property will indicate the last time a particular file was accessed, but this doesn't mean by a user. Many other processes and applications, such as anti-virus programs and defragmentation utilities, can affect this file property. Do not rely on this property as an indication of the last time a user accessed a file. You need to enable auditing to obtain that information.

PowerShell makes it very easy to change the value of any of these properties. This is similar to the **Touch** command.

```
(Get-ChildItem file.txt).Set_LastAccessTime("12/31/2014 01:23:45")
(Get-ChildItem file.txt).Set_LastWriteTime("12/31/2014 01:23:45")
(Get-ChildItem file.txt).Set_CreationTime("12/31/2014 01:23:45")
Get-ChildItem file.txt | select Name, CreationTime, LastWriteTime, LastAccessTime |
Format-Table -AutoSize
```

Here, we've set all the timestamps to "12/31/2014 01:23:45". Obviously, use this power with caution.

Another property that you can easily set from the console is file encryption. You accomplish this by using the **Encrypt()** method.

```
(Get-ChildItem file.txt).Attributes
Archive
```

```
(Get-ChildItem file.txt).Encrypt()
(Get-ChildItem file.txt).Attributes
Archive, Encrypted
```

You can still open the file because PowerShell encrypted it with your private key, but anyone else attempting to open the file will fail. To reverse the process, simply use the **Decrypt()** method.

```
(Get-ChildItem file.txt).Decrypt()
(Get-ChildItem file.txt).Attributes
Archive
```

## Proceed with caution

Before you get carried away and start encrypting everything—stop. Using the encrypting file system requires some serious planning and testing. You have to plan for recovery agents, lost keys, and more. This is not a task to undertake lightly. You need to research this topic and test thoroughly in a non-production environment.

What about compression? Even though compression can be indicated as a file attribute, you cannot compress a file simply by setting the attribute. Nor does the .NET file object have a compression method. The easy solution is to use Compact.exe to compress files and folders.

The other approach is to use WMI.

```
$wmofile = Get-WmiObject -Query "Select * from CIM_DATAFILE where name='c:\\file.txt'"
```

You would think all you need to do is set the **Compressed** property to **TRUE**.

```
$wmofile.Compressed=$TRUE
$wmofile.Compressed
True
```

It looks like it works, but when you examine the file in Windows Explorer, you'll see that it isn't actually compressed.

```
(Get-ChildItem file.txt).Attributes
Archive
```

You need to use the **Compress()** method.

```
$wmofile.Compress()
(Get-ChildItem file.txt).Attributes
Archive, Compressed
```

You can also confirm this in Windows Explorer. To reverse the process, use the **Uncompress()** method.

```
$wmofile.Uncompress()
```

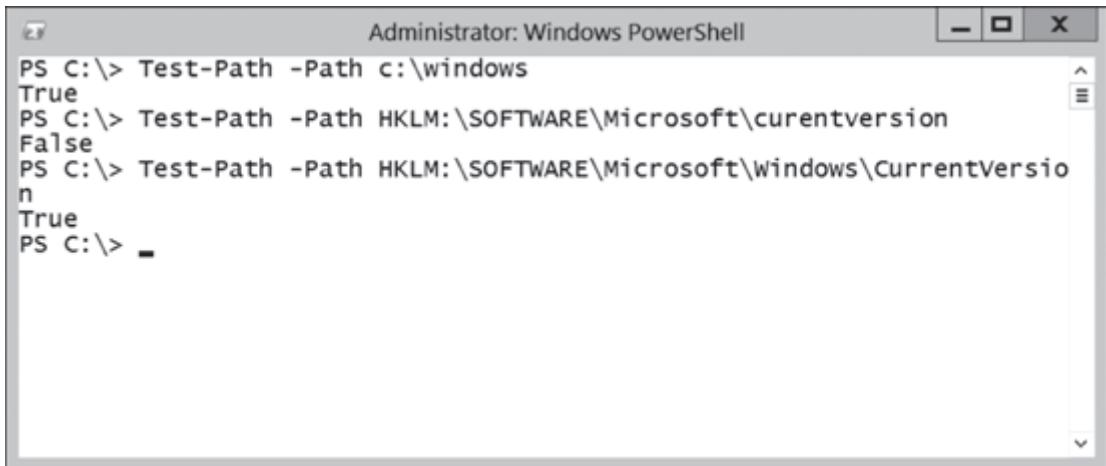
## Working with paths

When working with folders in PowerShell, you are also working with paths. A path, as the name suggests, is the direction to reach a particular destination. For a short and likely known path like C:\Windows, working with paths isn't critical. However, as your scripts grow in complexity or you are dealing with path variables, you'll want to be familiar with PowerShell's **Path** cmdlets. These cmdlets work with any provider that uses paths, such as the registry and certificate store.

### Test-Path

Adding error handling to a script is always helpful, especially when dealing with folders that may not exist. You can use the **Test-Path** cmdlet to validate the existence of a given path. The cmdlet returns **True** if the path exists.

```
Test-Path c:\windows
```



The screenshot shows an Administrator Windows PowerShell window. The command PS C:\> Test-Path -Path c:\windows is run, followed by PS C:\> Test-Path -Path HKLM:\SOFTWARE\Microsoft\currentversion and PS C:\> Test-Path -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion. The output for the first command is True, for the second is False, and for the third is True. The window has standard operating system window controls (minimize, maximize, close) and scroll bars on the right side.

```
Administrator: Windows PowerShell
PS C:\> Test-Path -Path c:\windows
True
PS C:\> Test-Path -Path HKLM:\SOFTWARE\Microsoft\currentversion
False
PS C:\> Test-Path -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion
True
PS C:\> _
```

Figure 2-13

You can also use this cmdlet to verify the existence of registry keys.

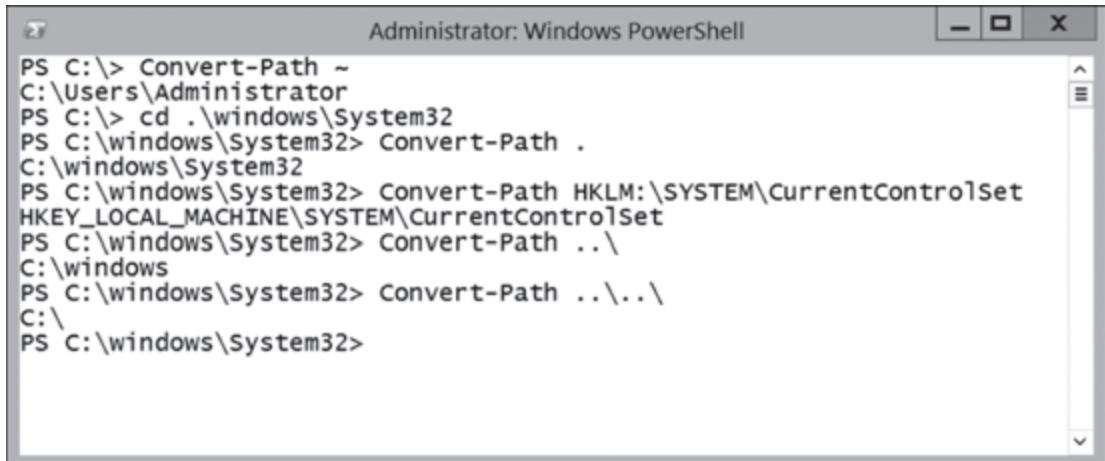
```
Test-Path HKLM:\Software\Microsoft\Windows\CurrentVersion
```

```
Test-Path HKLM:\Software\MyCompany\MyApp\Settings
```

### Convert-Path

Before PowerShell can work with paths, they also need to be properly formatted. The **Convert-Path** cmdlet will take paths or path variables and convert them to a format PowerShell can understand. For example, the ~ character represents your user profile path. Using the **Convert-Path** cmdlet will return the explicit path.

```
Convert-Path ~
```



The screenshot shows an 'Administrator: Windows PowerShell' window. The command 'Convert-Path ~' was run, followed by several other commands demonstrating path conversion:

```

PS C:\> Convert-Path ~
C:\Users\Administrator
PS C:\> cd .\windows\System32
PS C:\windows\System32> Convert-Path .
C:\windows\System32
PS C:\windows\System32> Convert-Path HKLM:\SYSTEM\CurrentControlSet
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet
PS C:\windows\System32> Convert-Path ..\
C:\windows
PS C:\windows\System32> Convert-Path ..\..\\
C:\
PS C:\windows\System32>

```

Figure 2-14

Here are some other ways you might use this cmdlet.

```
Convert-Path .
```

```
Convert-Path HKLM:\SYSTEM\CurrentControlSet
```

```
Convert-Path ..\
```

```
Convert-Path ..\..\
```

You could also achieve the last two examples by using the **Split-Path** cmdlet.

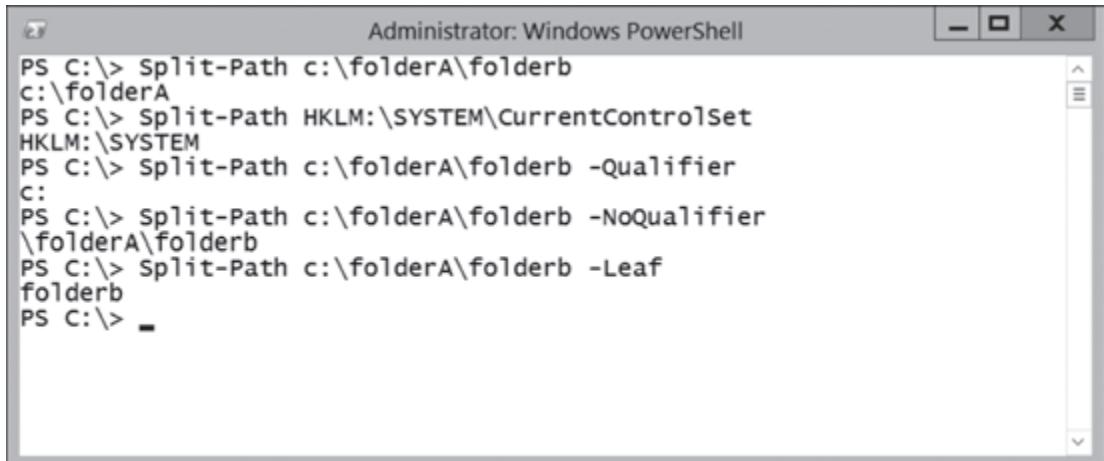
## Split-Path

The **Split-Path** cmdlet will split a given path into its parent or leaf components. You can use this cmdlet to display or reference different components of a given path. The default is to split the path and return the parent path component.

```
Split-Path c:\folderA\FolderB
```

This cmdlet also works with other providers, such as the registry.

```
Split-Path HKLM:\SYSTEM\CurrentControlSet
```



The screenshot shows an Administrator Windows PowerShell window with the title "Administrator: Windows PowerShell". The command history is as follows:

```
PS C:\> Split-Path c:\folderA\folderb
c:\folderA
PS C:\> Split-Path HKLM:\SYSTEM\CurrentControlSet
HKLM:\SYSTEM
PS C:\> Split-Path c:\folderA\folderb -Qualifier
c:
PS C:\> Split-Path c:\folderA\folderb -NoQualifier
\folderA\folderb
PS C:\> Split-Path c:\folderA\folderb -Leaf
folderb
PS C:\> _
```

Figure 2-15

The **Split-Path** cmdlet will include the specified path's qualifier, which is the provider path's drive such as D:\ or HKLM:. If you wanted to find just the qualifier, in essence the root, use the **-Qualifier** parameter.

```
Split-Path C:\folderA\FolderB -Qualifier
```

You can use the **-NoQualifier** parameter to return a path without the root.

```
Split-Path C:\folderA\FolderB -NoQualifier
```

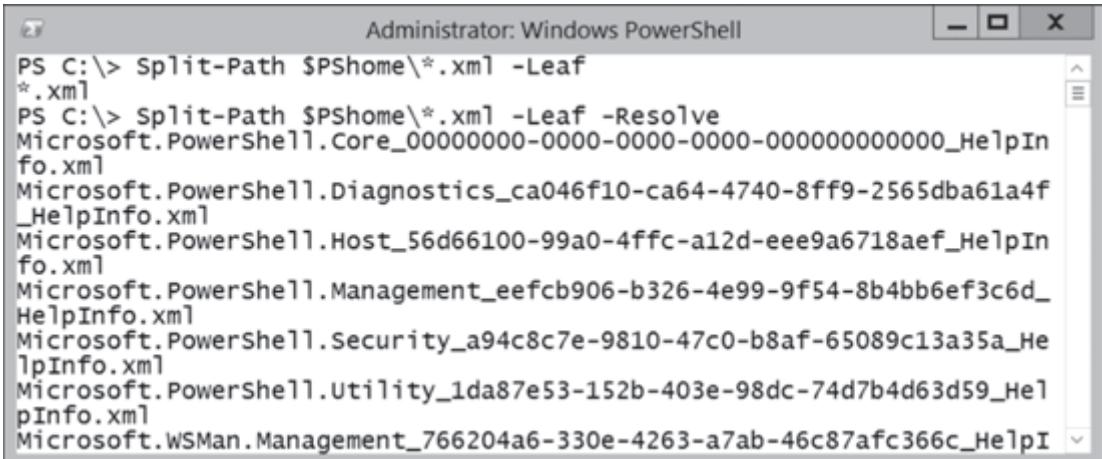
```
Split-Path HKLM:\SYSTEM\CurrentControlSet -NoQualifier
```

However, if you need the last part of the path and not the parent, use the **-Leaf** parameter.

```
Split-Path c:\folderA\FolderB -Leaf
```

When using path variables, you may need to include the **-Resolve** parameter to expand the variable to its full path.

```
Split-Path $pshome\*.xml -Leaf -Resolve
```

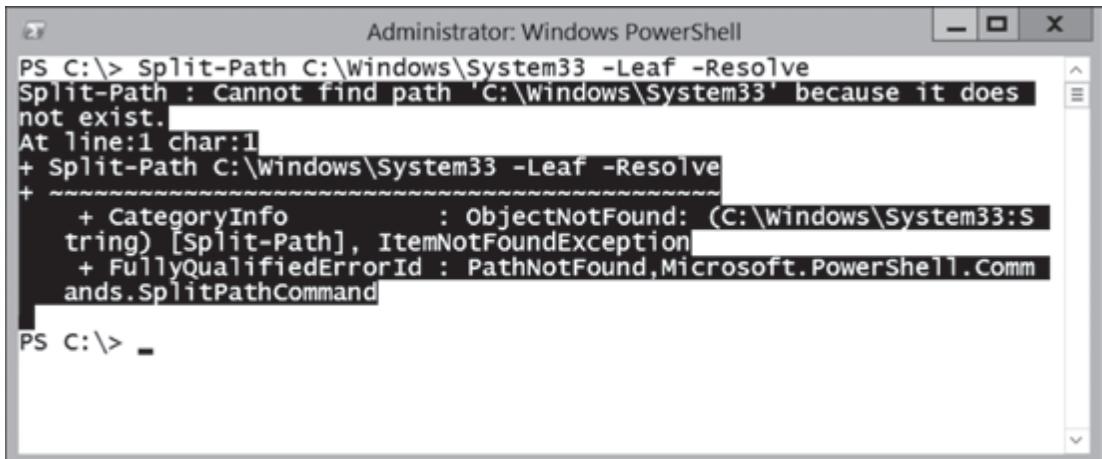


```
Administrator: Windows PowerShell
PS C:\> Split-Path $PSHome\*.xml -Leaf
*.xml
PS C:\> Split-Path $PSHome\*.xml -Leaf -Resolve
Microsoft.PowerShell.Core_00000000-0000-0000-0000-000000000000_HelpIn
fo.xml
Microsoft.PowerShell.Diagnostics_ca046f10-ca64-4740-8ff9-2565dba61a4f
_HelpInfo.xml
Microsoft.PowerShell.Host_56d66100-99a0-4ffc-a12d-eee9a6718aef_HelpIn
fo.xml
Microsoft.PowerShell.Management_eefcb906-b326-4e99-9f54-8b4bb6ef3c6d_
HelpInfo.xml
Microsoft.PowerShell.Security_a94c8c7e-9810-47c0-b8af-65089c13a35a_He
lplibInfo.xml
Microsoft.PowerShell.Utility_1da87e53-152b-403e-98dc-74d7b4d63d59_Hel
pInfo.xml
Microsoft.WSMAN.Management_766204a6-330e-4263-a7ab-46c87afc366c_HelpI
```

Figure 2-16

This parameter is also useful in confirming that a path component exists.

```
Split-Path C:\Windows\System33 -Leaf -Resolve
```



```
Administrator: Windows PowerShell
PS C:\> Split-Path C:\Windows\System33 -Leaf -Resolve
Split-Path : Cannot find path 'C:\Windows\System33' because it does
not exist.
At line:1 char:1
+ Split-Path C:\Windows\System33 -Leaf -Resolve
+ ~~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\Windows\System33:S
tring) [Split-Path], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Com
ands.SplitPathCommand
PS C:\> _
```

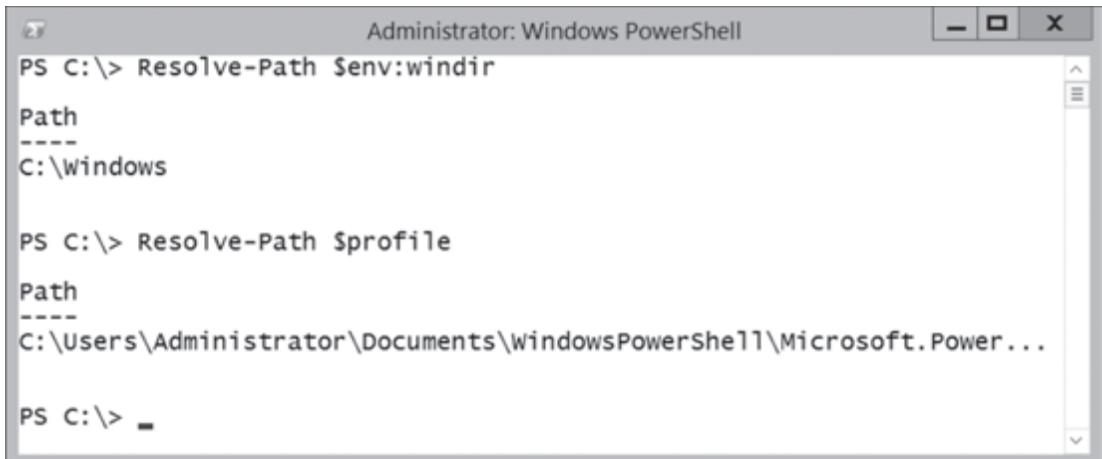
Figure 2-17

PowerShell also has a specific cmdlet for resolving path names.

## Resolve-Path

The **Resolve-Path** cmdlet is primarily intended to resolve wildcards or variables that might be part of a path.

```
Resolve-Path $env:windir
```



```
Administrator: Windows PowerShell
PS C:\> Resolve-Path $env:windir
Path
-----
C:\Windows

PS C:\> Resolve-Path $profile
Path
-----
C:\Users\Administrator\Documents\WindowsPowerShell\Microsoft.Power...  

PS C:\> -
```

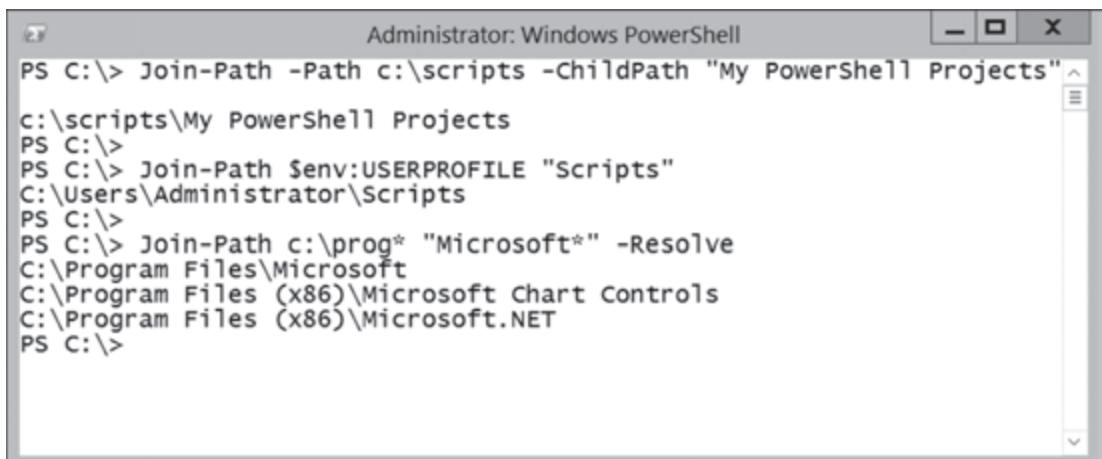
Figure 2-18

Occasionally, you need to construct a path from disparate elements, and the **Join-Path** cmdlet will create a PowerShell-ready path.

```
Join-Path -Path c:\scripts -Childpath "My PowerShell Projects"
```

The **-Path** and **-ChildPath** parameters are positional, so you don't need to specify the parameter names. The following example will give you the same result as the previous example.

```
Join-Path c:\scripts "My PowerShell Projects"
```



```
Administrator: Windows PowerShell
PS C:\> Join-Path -Path c:\scripts -ChildPath "My PowerShell Projects"
c:\scripts\My PowerShell Projects
PS C:\>
PS C:\> Join-Path $env:USERPROFILE "Scripts"
C:\Users\Administrator\Scripts
PS C:\>
PS C:\> Join-Path c:\prog* "Microsoft*" -Resolve
C:\Program Files\Microsoft
C:\Program Files (x86)\Microsoft Chart Controls
C:\Program Files (x86)\Microsoft .NET
PS C:\>
```

Figure 2-19

The path name doesn't have to be explicit. You can also use variables.

```
Join-Path $env:USERPROFILE "Scripts"
```

The cmdlet can also work with wildcards and the **-Resolve** parameter, to expand the wildcard and create a complete path.

```
Join-Path c:\prog* "microsoft*" -Resolve
```

## Creating directories

Creating directories in PowerShell is nearly the same as it is in Cmd.exe.

```
mkdir "NewFiles"
```

Alternatively, you can use the **New-Item** cmdlet, which offers a few more features.

```
New-Item -Path c:\testdir -ItemType directory
```

The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. It contains two command-line entries:

```
PS C:\> New-Item -Path c:\testdir -ItemType directory
Directory: C:\

Mode          LastWriteTime      Length Name
----          -----          ----- 
d---  7/1/2013 11:17 AM          testdir

PS C:\> New-Item -Path \\server\c$\testdir -ItemType directory
Directory: \\server\c$
```

Figure 2-20

The cmdlet also creates nested directories. In other words, like mkdir in Cmd.exe, it creates any intermediate directories.

```
New-Item -ItemType directory c:\temp\1\2\3
```

## Listing directories

Even though you can use **dir** in PowerShell to list directories and files, it is really an alias for the **Get-ChildItem** cmdlet. However, you can specify files to include or exclude in the search and also recurse through subdirectories.

```
dir -Exclude *.old -Recurse
```

Remember: even though PowerShell is displaying text, it is really working with objects. This means you can be creative in how you display information.

```
dir -Exclude *.old,*.bak,*.tmp -Recurse | Select-Object ` 
    FullName, Length, LastWriteTime | Format-Table -AutoSize
```

This expression recurses from the starting directory and lists all files that don't end in .old, .bak, or .tmp. Using the **dir** alias, we piped the output from the **Get-ChildItem** cmdlet to the **Select-Object** cmdlet, because we wanted to display only certain information formatted as a table.

## Deleting directories

Deleting a directory is essentially the same as deleting a file. You can use the **rmdir** alias for the **Remove-Item** cmdlet.

```
Remove-Item -Path c:\temp
```

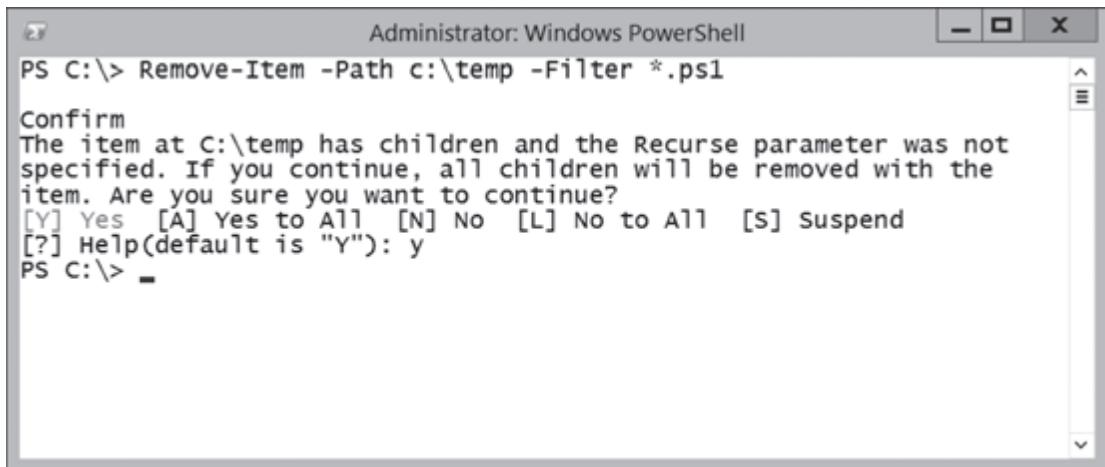


Figure 2-21

PowerShell gives you a warning if you attempt to remove a directory that isn't empty.



## In Depth 03

# Managing permissions

## Managing permissions

Managing file permissions with scripting has always been a popular and challenging task. Even though PowerShell provides new ways to access and work with access-control lists (ACLs), you still may find more familiar command-line utilities—such as **Cacls.exe**, **Xcacls.exe**, **Dsacls.exe**—easier to use. And the good news is that you can use them right from within PowerShell! In this chapter, we'll also look at PowerShell's native abilities to work with permissions.

## Viewing permissions

You can use the **Get-Acl** cmdlet to obtain security descriptor information for files, folders, printers, registry keys, and more. By default, all information is displayed in a table format.

```
get-acl $env:systemroot\regedit.exe
```

```

Administrator: Windows PowerShell
PS C:\> get-acl $env:systemroot\regedit.exe

Directory: C:\Windows

Path          Owner          Access
----          -----          -----
regedit.exe   NT SERVICE\TrustedI...  NT AUTHORITY\SYSTEM...

```

PS C:\>

Figure 3-1

The problem is that some of the information is truncated. Therefore, you'll probably prefer to use something like the following:

```
get-acl $env:systemroot\regedit.exe | Format-list
```

```

Administrator: Windows PowerShell
PS C:\> get-acl $env:systemroot\regedit.exe | Format-list

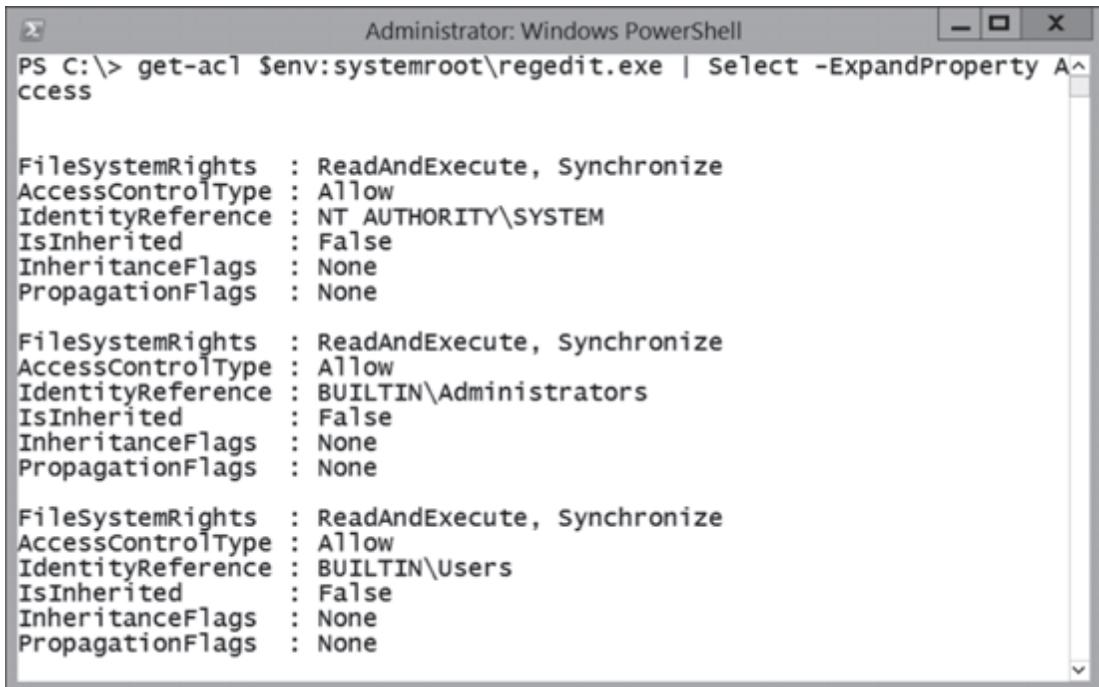
Path      : Microsoft.PowerShell.Core\FileSystem::C:\Windows\regedit.exe
Owner     : NT SERVICE\TrustedInstaller
Group    : NT SERVICE\TrustedInstaller
Access   : NT AUTHORITY\SYSTEM Allow ReadAndExecute, Synchronize
           BUILTIN\Administrators Allow ReadAndExecute, Synchronize
           BUILTIN\Users Allow ReadAndExecute, Synchronize
           NT SERVICE\TrustedInstaller Allow FullControl
           APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES
           Allow ReadAndExecute, Synchronize
Audit    :
Sddl     : O:S-1-5-80-956008885-3418522649-1831038044-1853292631-227147
           8464G:S-1-5-80-956008885-3418522649-1831038044-1853292631-22
           71478464D:PAI(A;;0x1200a9;;;SY)(A;;0x1200a9;;;BA)(A;;0x1200a

```

Figure 3-2

The **Get-Acl** cmdlet writes an object to the pipeline that is part of the **System.Security.AccessControl** .NET class. Access information is presented as a **System.Security.AccessControl.FileSystemAccessRule** object, which you can view by using the **Access** property.

```
get-acl $env:systemroot\regedit.exe | Select -ExpandProperty Access
```



```
Administrator: Windows PowerShell
PS C:\> get-acl $env:systemroot\regedit.exe | Select -ExpandProperty Access

FileSystemRights : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited : False
InheritanceFlags : None
PropagationFlags : None

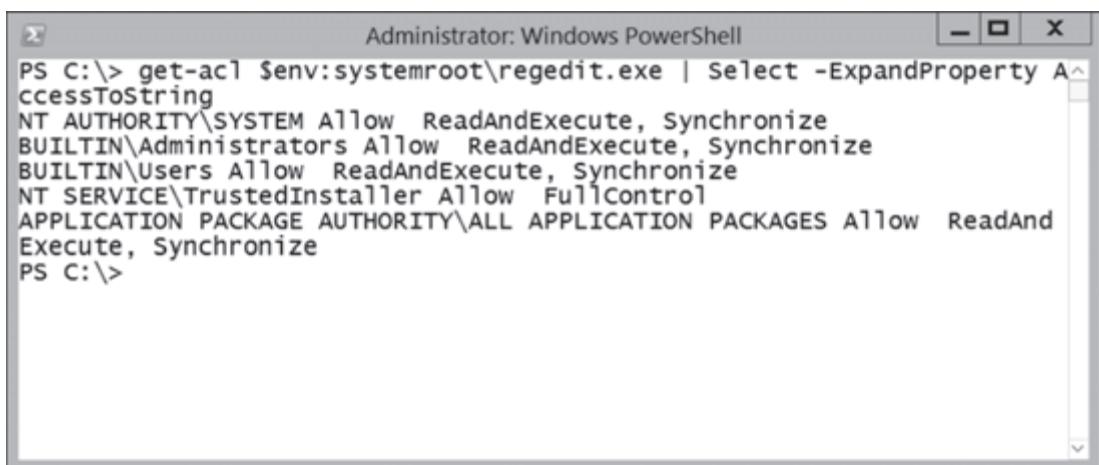
FileSystemRights : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited : False
InheritanceFlags : None
PropagationFlags : None

FileSystemRights : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : BUILTIN\Users
IsInherited : False
InheritanceFlags : None
PropagationFlags : None
```

Figure 3-3

Or, you may prefer output that is more like Cacls.exe, which you can view by using the **AccessToString** property.

```
get-acl $env:systemroot\regedit.exe | Select -ExpandProperty AccessToString
```



```
Administrator: Windows PowerShell
PS C:\> get-acl $env:systemroot\regedit.exe | Select -ExpandProperty AccessToString
NT AUTHORITY\SYSTEM Allow ReadAndExecute, Synchronize
BUILTIN\Administrators Allow ReadAndExecute, Synchronize
BUILTIN\Users Allow ReadAndExecute, Synchronize
NT SERVICE\TrustedInstaller Allow FullControl
APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES Allow ReadAnd
Execute, Synchronize
PS C:\>
```

Figure 3-4

The **Get-Acl** cmdlet also works for directories.

```
get-acl $pshome | format-list
```

```
Path      : Microsoft.PowerShell.Core\FileSystem::C:\windows\System32\WindowsPowerShell\v1.0
Owner     : NT SERVICE\TrustedInstaller
Group    : NT SERVICE\TrustedInstaller
Access   : CREATOR OWNER Allow 268435456
           NT AUTHORITY\SYSTEM Allow 268435456
           NT AUTHORITY\SYSTEM Allow Modify, Synchronize
           BUILTIN\Administrators Allow 268435456
           BUILTIN\Administrators Allow Modify, Synchronize
           BUILTIN\Users Allow -1610612736
           BUILTIN\Users Allow ReadAndExecute, Synchronize
           NT SERVICE\TrustedInstaller Allow 268435456
           NT SERVICE\TrustedInstaller Allow FullControl
```

Figure 3-5

It will even work on registry keys.

```
get-acl HKLM:\software\microsoft\windows\CurrentVersion\run | Format-List
```

```
Path      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software\microsoft\windows\CurrentVersion\run
Owner     : NT AUTHORITY\SYSTEM
Group    : NT AUTHORITY\SYSTEM
Access   : BUILTIN\Users Allow ReadKey
           BUILTIN\Users Allow -2147483648
           BUILTIN\Administrators Allow FullControl
           BUILTIN\Administrators Allow 268435456
           NT AUTHORITY\SYSTEM Allow FullControl
           NT AUTHORITY\SYSTEM Allow 268435456
           CREATOR OWNER Allow 268435456
           APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES
```

Figure 3-6

Notice that the cmdlet returns the owner. You can create a **Get-Acl** expression to display just that information.

```
get-acl c:\scripts\*.ps1 | format-table PSChildName, Owner -autosize
```

```
Administrator: Windows PowerShell
PS C:\> Get-acl c:\scripts\*.ps1 | format-table PSChildName,Owner -autosize

PSChildName          Owner
-----            -----
DiskInfo.ps1        BUILTIN\Administrators
Find-USer.ps1       BUILTIN\Administrators
Get-Required.ps1    BUILTIN\Administrators
GetEvent.ps1         BUILTIN\Administrators
GetProcess.ps1      BUILTIN\Administrators
GetService.ps1      BUILTIN\Administrators
ShowControlPanelItem.ps1 BUILTIN\Administrators
test.ps1            BUILTIN\Administrators

PS C:\>
```

Figure 3-7

## Viewing permissions for an entire object hierarchy

The **Get-Acl** cmdlet doesn't have a **Recurse** parameter, but that won't slow us down. **Get-Acl** does support pipeline input from **Get-ChildItem**. If you want a report to show owners for a directory structure, you can use something like the example in Figure 3-8.

```
Get-ChildItem c:\scripts -Recurse | Get-Acl | Select Path, Owner | Format-List
```

```
Administrator: Windows PowerShell
PS C:\> Get-ChildItem c:\scripts -Recurse | Get-Acl | Select Path, Owner | Format-List

Path   : Microsoft.PowerShell.Core\FileSystem::C:\scripts\MyNewFolder
Owner  : BUILTIN\Administrators

Path   : Microsoft.PowerShell.Core\FileSystem::C:\scripts\subfolder
Owner  : BUILTIN\Administrators

Path   : Microsoft.PowerShell.Core\FileSystem::C:\scripts\DiskInfo.ps1
Owner  : BUILTIN\Administrators

Path   : Microsoft.PowerShell.Core\FileSystem::C:\scripts\Find-USer.ps
1
Owner  : BUILTIN\Administrators
```

Figure 3-8

The **Get-ChildItem** cmdlet passes every item to the **Get-Acl** cmdlet and recurses through subdirectories. You'll notice that we piped output to the **Select-Object** cmdlet, to receive just the **Path** and **Owner** properties.

```
Get-ChildItem c:\scripts -Recurse | Get-Acl | Select Path, Owner | Export-Csv C:\report.csv
-NoTypeInformation
```

Finally, you can send the data to a CSV or to an XML file by using **Export-CliXML** for the finished report.

## Changing permissions

Retrieving ACLs is half the job. You might still want to reset permissions through PowerShell. To be honest, this is not the easiest task to do in PowerShell, mainly because permissions in Windows are complicated, and there's only so much a shell can do to simplify that situation.

To do really detailed work with permissions, you need to understand .NET security objects and NTFS security descriptors. However, we'll start with some simpler examples. Setting an access-control rule is a matter of bit-masking access rights against a security token. The bits that match a security principal's account determine whether you can view a file, make changes to a file, or take ownership.

You can use the **Set-Acl** cmdlet to update an object's access rule. However, you first have to construct a .NET security descriptor or retrieve the security descriptor from an existing object, modify the security descriptor appropriately, and then apply it to the desired object. This is not an insurmountable task—just very tedious. The script **ChangeACL.ps1** takes a simplified approach and grants permissions you specify to the specified security principal on all objects in the specified starting directory and subdirectories.

### ChangeACL.ps1

```
$Right="FullControl"

#The possible values for Rights are
# ListDirectory
# ReadData
# WriteData
# CreateFiles
# CreateDirectories
# AppendData
# ReadExtendedAttributes
# WriteExtendedAttributes
# Traverse
# ExecuteFile
# DeleteSubdirectoriesAndFiles
# ReadAttributes
# WriteAttributes
# Write
# Delete
# ReadPermissions
# Read
# ReadAndExecute
# Modify
# ChangePermissions
# TakeOwnership
# Synchronize
# FullControl
```

```
$StartingDir=Read-Host " What directory do you want to start at?"
$Principal=Read-Host " What security principal do you want to grant" `n
"$Right to? `n Use format domain\username or domain\group"

#define a new access rule
$rule = New-Object System.Security.AccessControl.FileSystemAccessRule($Principal, $Right, "Allow")

foreach ($file in $(Get-ChildItem $StartingDir -Recurse)) {
    $acl = Get-Acl $file.FullName
    #display filename and old permissions
    Write-Host -Foregroundcolor Yellow $file.FullName
    #uncomment if you want to see old permissions
    #Write-Host $acl.ToString `n

    #Add this access rule to the ACL
    $acl.SetAccessRule($rule)

    #Write the changes to the object
    Set-Acl $File.FullName $acl

    #display new permissions
    $acl = Get-Acl $file.FullName
    Write-Host -Foregroundcolor Green "New Permissions"
    Write-Host $acl.ToString `n
} #end foreach file
```

This script creates a simple access rule that allows a specific right. If you can use a broad right, such as **Modify** or **Full Control**, you'll find it easy to work with the script. We've hard coded the **\$Right** variable. The script prompts you for the directory path and the name of the security principal to which you wish to apply the right.

The real work of the script is creating a new **FileSystemAccess** rule object. Creating the object requires that we specify the name of the security principal, the right to be applied, and whether to allow or deny the right. With this rule, we can recursively go through the file system, starting at the specified directory. For each file, we receive the current access-control list, by using the **Get-Acl** cmdlet.

```
$acl = Get-Acl $file.FullName
```

Next we add the new access control rule to the ACL.

```
$acl.SetAccessRule($rule)
```

Now we call the **Set-Acl** cmdlet to write the new and modified ACL back to the object.

```
Set-Acl $File.FullName $acl
```

The script finishes the loop by displaying the new ACL so you can see the changes.

## Automating Cacls.exe to change permissions

As you've seen, using the **Set-Acl** cmdlet is not simple, especially if you have complex permissions. Therefore, you may find it easier to use Cacls.exe from within a PowerShell script.

### SetPermswithCACLS.ps1

```
#SetPermsWithCACLS.ps1
# CACLS rights are usually
# F = FullControl
# C = Change
# R = Readonly
# W = Write

$StartingDir = Read-Host " What directory do you want to start at?"
$Right = Read-Host " What CALCS right do you want to grant? Valid choices are F, C, R or W"
Switch ($Right) {
    "F" {$Null}
    "C" {$Null}
    "R" {$Null}
    "W" {$Null}
    default {
        Write-Host -ForegroundColor "Red" `n
        `n $Right.ToUpper() "is an invalid choice. Please Try again."`n
        exit
    }
}

$Principal = Read-Host " What security principal do you want to grant" `n
"CACLS right"$Right.ToUpper()"to?" `n
"Use format domain\username or domain\group"

$Verify = Read-Host `n "You are about to change permissions on all" `n
"files starting at"$StartingDir.ToUpper() `n "for security" `n
"principal"$Principal.ToUpper() `n
"with new right of"$Right.ToUpper()." `n
"Do you want to continue ? [Y,N]"

if ($Verify -eq "Y") {

    foreach ($file in $(Get-ChildItem $StartingDir -Recurse)) {
        #display filename and old permissions
        Write-Host -ForegroundColor Yellow $file.FullName
        #uncomment if you want to see old permissions
        #CACLS $file.FullName

        #ADD new permission with CACLS
        CACLS $file.FullName /E /P "${Principal}: ${Right}" >$NULL
        #display new permissions
        Write-Host -ForegroundColor Green "New Permissions"
        CACLS $file.FullName
    }
}
```

This script first prompts you for a starting directory and the permission rights you want to grant. We've used a **Switch** statement to make sure a valid parameter for Cacls.exe is entered. As long as you entered **F**, **C**, **W**, or **R**, the script continues and prompts you for the name of a security principal that you want to add to the ACL. Since this is a major operation, we've included a prompt by using the **Read-Host** cmdlet to provide a summary of what the script is about to do. If anything other than **Y** is entered, the script ends, with no changes being made. Otherwise, the script executes the **Foreach** loop.

Within this **Foreach** loop, we use the **Get-ChildItem** cmdlet to enumerate all the files in the starting directory path and recurse through all subdirectories. The script displays the current file as a progress indicator, and then calls Cacls.exe. Due to the way PowerShell processes Win32 commands such as Cacls.exe, we need to enclose the program's parameters in quotes. You'll also notice that instead of using the following:

```
CACLS $file.FullName /e /p "$Principal:$Right"
```

That we used a modified version, instead.

```
CACLS $file.FullName /e /p "${Principal}: ${Right}"
```

PowerShell treats an expression like **Foo:Bar** as **<namespace>:<name>**, which is like **\$global:profile** or **\$env:windir**. In order for PowerShell to treat the Cacls.exe parameter as a command-line parameter, we must delimit the variable name, by using braces, as we did in the previous example. The script finishes by displaying the new access-control permissions for each file.

If you've used Cacls.exe before, you may have noticed that we used **/E /P** to assign permissions. According to Cacls.exe's Help screen, you use **/P** to modify permissions for an existing entry. You would use **/G** to grant permissions to a new user. In Cmd.exe, either **/G** or **/P** will work, regardless of whether the user already existed in the access-control list.

This is not the case in PowerShell. PowerShell actually appears to enforce the Cacls.exe parameters. You can use **/G** if a user does not exist in the file's access-control list. However, you must use **/P** if the user already exists. When you attempt to use **/G** to modify an existing user's permission, Cacls.exe will run, but no change will be made.

So, how do you know if you should use **/P** or **/G**, without checking every file first? Not to worry. You can use **/P** regardless of whether the user exists in the access-control list, which is what we've done here. The moral is, don't assume that every Cmd.exe tool and command works identically in PowerShell. Most should, but if it doesn't, you have to look at how PowerShell is interpreting the expression.

One final note about the script: we could have used **/T** with Cacls.exe to change permissions on all files and subdirectories. The end result would have been the same, but then we couldn't have demonstrated some of PowerShell's output features.

## Complex permissions in PowerShell

By now, you've seen that you can manage permissions with PowerShell, although it is not for the faint of heart. That said, let's look at a few more situations where you can use PowerShell.

### Get-Owner

We showed you earlier how you can use the **Get-Acl** cmdlet to display the owner of a file. You may prefer to create a function that takes a file name as a parameter.

```
Function Get-Owner {
param([string]$file)

try {
Get-Item $file -ea stop | Out-Null
write (Get-Acl $file).Owner
}
catch {
Write-Warning "Failed to find $file"
}
finally {}
}
```

With this function loaded, you can use an expression like the one in Figure 3-9.

```
Get-Owner c:\file.txt
```

The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command PS C:\> Function Get-Owner { ... } is entered and displayed. Below it, the command PS C:\> get-owner C:\scripts\DiskInfo.ps1 is entered and executed, resulting in the output 'BUILTIN\Administrators'.

```
Administrator: Windows PowerShell
PS C:\> Function Get-Owner {
>> param([string]$file)
>>
>> try {
>> Get-Item $file -ea stop | Out-Null
>> write (Get-Acl $file).Owner
>>
>> catch {
>> Write-Warning "Failed to find $file"
>> }
>> finally {}
>> }
>>
PS C:\> get-owner C:\scripts\DiskInfo.ps1
BUILTIN\Administrators
PS C:\>
```

Figure 3-9

To return the owners on a group of files, you have to enumerate the file collection.

```
dir c:\scripts | select fullname, @{name="Owner"; Expression = { Get-Owner $_.fullname } }
```

```
Administrator: Windows PowerShell
PS C:\> dir c:\scripts | select fullname,@{name="Owner";Expression={Get-Owner $_.fullname}}
FullName                               Owner
-----
C:\scripts\MyNewFolder                BUILTIN\Administrators
C:\scripts\subfolder                  BUILTIN\Administrators
C:\scripts\DiskInfo.ps1               BUILTIN\Administrators
C:\scripts\Find-User.ps1              BUILTIN\Administrators
C:\scripts\Get-Required.ps1           BUILTIN\Administrators
C:\scripts\GetEvent.ps1               BUILTIN\Administrators
C:\scripts\GetProcess.ps1             BUILTIN\Administrators
C:\scripts\GetService.ps1             BUILTIN\Administrators
C:\scripts\HelpFullNotes.txt          BUILTIN\Administrators
C:\scripts\scripts.recall            BUILTIN\Administrators
C:\scripts>ShowControlPanelItem...    BUILTIN\Administrators
```

Figure 3-10

Since we are leveraging the pipeline, you can accomplish something like the following:

```
dir c:\scripts -Recurse | select fullname, @{name="Owner"; Expression={Get-Owner $_.fullname}} | Group-Object owner | Format-Table Name, Count -AutoSize
```

## SetOwner

Unfortunately, PowerShell does not provide a mechanism for setting the owner of a file to anything other than an administrator or the Administrators group. Even though Windows 2003 and later allows you to assign ownership, you cannot do it through PowerShell. Here's how to set a new owner on a file.

```
[System.Security.Principal.NTAccount]$newOwner = "Administrators"
$file = Get-ChildItem c:\file.txt
$acl = $file.GetAccessControl()
$acl.SetOwner($newOwner)
$file.SetAccessControl($acl)
```

The important step is to cast the **\$NewOwner** variable as a security principal.

```
[System.Security.Principal.NTAccount]$newOwner = "Administrators"
```

After we retrieve the current access-control list, we call the **SetOwner()** method and specify the new owner.

```
$acl.SetOwner($newOwner)
```

This change will not take effect until we call the **SetAccessControl()** method on the file and apply the modified access-control list with the new owner.

```
$file.SetAccessControl($acl)
```

## Retrieving access control

We showed you at the beginning of the chapter how to use the **Get-Acl** cmdlet to retrieve file permissions. One approach you might take is to wrap the code into a function.

```
Function Get-Access {
    param([string]$file)
    write $file.ToUpper()
    write (Get-Acl $file).Access
}
```

Once you've loaded it into your PowerShell session or script, you can use it to quickly find the access control for a given file.

```
Get-Access $env:systemroot\system32\cmd.exe
```

```
Administrator: Windows PowerShell
PS C:\> Get-Access $env:systemroot\system32\cmd.exe
C:\WINDOWS\SYSTEM32\CMD.EXE

FileSystemRights : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited : False
InheritanceFlags : None
PropagationFlags : None

FileSystemRights : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited : False
InheritanceFlags : None
```

Figure 3-11

The downside to this approach is that you can't easily use it in the pipeline. An expression like the following will fail to enumerate all the files.

```
Get-ChildItem c:\files\*.txt | Get-Access | Format-Table
```

A better approach is to use an advanced function.

```
Function Get-AccessControl {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory = $true,
        ValueFromPipeline = $true,
```

```

        ValueFromPipelineByPropertyName = $true)]
[string[]]$file
)

Process{
    Foreach($item in $file){
        $access = Get-Acl -Path $item

        $prop=@{
            'Owner' = $access.Owner;
            'Access' = $access.AccessToString
        }
        $obj = New-Object -TypeName PSObject -Property $Prop
        Write-Output $obj
    }
}
}

```

This is a simple starter function that you can add too. We've used the **New-Object** cmdlet to create a custom object to return file and access information. Since the **AccessControl** property of our custom object is a collection of access rules, you need to use an expression like the following, in order to expand them.

```
(Get-ChildItem c:\file.txt | Get-AccessControl).AccessControl
```

Or you can examine a group of files.

```
dir c:\files | foreach {
    Write-Host $_.fullname -Foregroundcolor Green
    ($_.fullname | Get-AccessControl).AccessControl
}
```

But what if you want to find a particular security principal? Use the following function to enumerate the **AccessControl** property and search for the particular user or group.

### **Get-PrincipalFilter.ps1**

```
Function Get-Principal {
Param([string]$Principal)

foreach ($rule in $_.AccessControl) {
    if ($rule.IdentityReference -eq $Principal) {
        $_.filename, $rule
    } #end if
} #end foreach
} #end filter
```

Use the following filter in conjunction with the **Get-AccessControl** cmdlet to display files and access rules that apply to a given user.

```
dir c:\files\*.exe | Get-Accesscontrol | Get-Principal "mycompany\file admins
```

## Removing a rule

Removing access for a user or group is relatively straightforward.

```
$file = "file.txt"
[System.Security.Principal.NTAccount]$principal = "mycompany\rgbiv"
$acl = Get-Acl $file
$access = (Get-Acl $file).Access
$rule = $access | where { $_.IdentityReference -eq $principal }
$acl.RemoveAccessRuleSpecific($rule)
Set-Acl $file $acl
```

Obviously, we need to know what file and user or group we are working with.

```
$file = "file.txt"
[System.Security.Principal.NTAccount]$principal = "mycompany\rgbiv"
```

As we did when adding a rule, we need to use the **Get-Acl** cmdlet to retrieve the current access control list.

```
$acl = Get-Acl $file
```

To find a specific access-control rule, we need to filter the existing rules with the **Where-Object** cmdlet.

```
$access = (Get-Acl $file).Access
$rule = $access | where { $_.IdentityReference -eq $principal }
```

The **\$rule** variable will hold all the rules that apply to the specified security principal. To remove the rule, we call the **RemoveAccessRuleSpecific()** method.

```
$acl.RemoveAccessRuleSpecific($rule)
```

Finally, to apply the new access control list, we call **Set-Acl**.

```
Set-Acl $file $acl
```

## In Depth 04

# Managing services

### Managing services

PowerShell offers several cmdlets that make managing Windows servers a breeze. The following sections will briefly discuss the cmdlets that you will use most often.

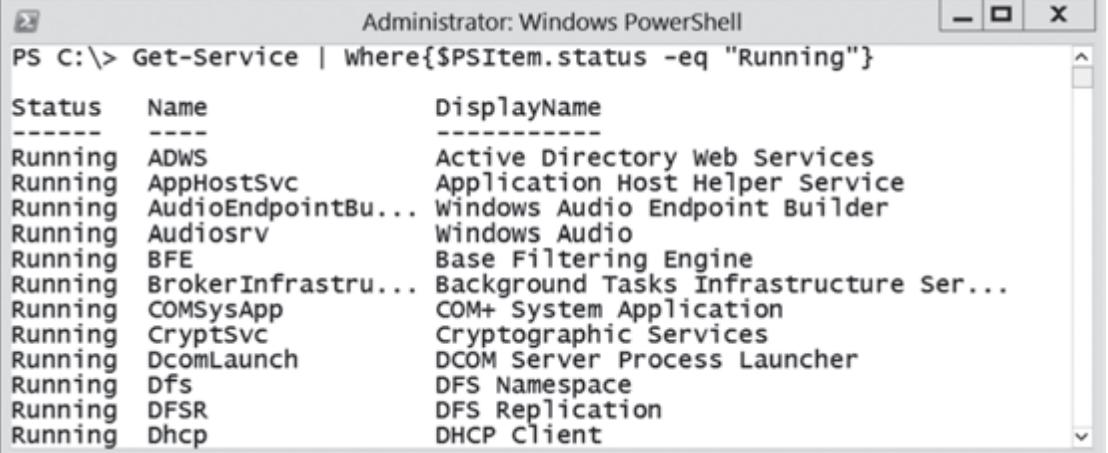
### Listing services

We have used the **Get-Service** cmdlet in many examples throughout this book. The **Get-Service** cmdlet, with no parameters, will display some information about the services on your computer. Usually you want to filter out services. The following is a standard expression to list all running services.

```
Get-Service | Where { $_.status -eq "Running" }
```

You can also use the new replacement variable for `$_`, which started with PowerShell version 3, called **\$PSItem**. You should familiarize yourself with both, as you will see both in examples on the Internet.

```
Get-Service | Where { $PSItem.status -eq "Running" }
```



```
Administrator: Windows PowerShell
PS C:\> Get-Service | Where{$PSItem.Status -eq "Running"}
```

Status	Name	DisplayName
Running	ADWS	Active Directory Web Services
Running	AppHostSvc	Application Host Helper Service
Running	AudioEndpointBu...	Windows Audio Endpoint Builder
Running	Audiosrv	Windows Audio
Running	BFE	Base Filtering Engine
Running	BrokerInfrastru...	Background Tasks Infrastructure Ser...
Running	COMSysApp	COM+ System Application
Running	CryptSvc	Cryptographic Services
Running	DcomLaunch	DCOM Server Process Launcher
Running	Dfs	DFS Namespace
Running	DFSR	DFS Replication
Running	Dhcp	DHCP Client

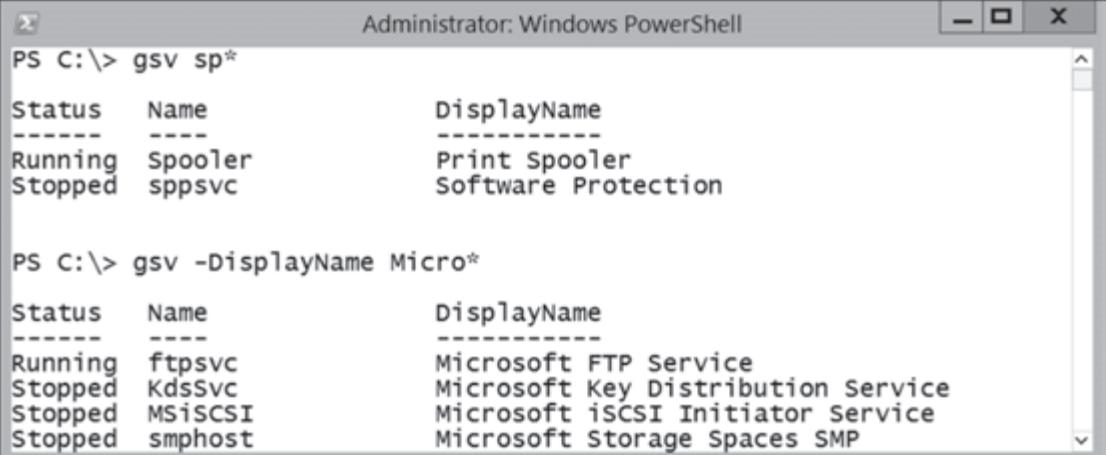
Figure 4-1

The **Get-Service** cmdlet also supports wildcards, which are useful when you aren't sure of the exact service name. The cmdlet's alias is **gsv**.

```
gsv sp*
```

Or if you only know services by their display name, the **-DisplayName** parameter will come in handy. It, too, supports wildcards.

```
gsv -DisplayName Micro*
```



```
Administrator: Windows PowerShell
PS C:\> gsv sp*
Status      Name           DisplayName
----      ----           -----
Running    Spooler        Print Spooler
Stopped   sppsvc         Software Protection

PS C:\> gsv -DisplayName Micro*
Status      Name           DisplayName
----      ----           -----
Running    ftpsvc         Microsoft FTP Service
Stopped   KdsSvc          Microsoft Key Distribution Service
Stopped   MSISCSI         Microsoft iSCSI Initiator Service
Stopped   smphost         Microsoft Storage Spaces SMP
```

Figure 4-2

PowerShell version 2 also made it easier to identify required services, that is, services that the specified service relies on.

```
gsv -name Winrm -RequiredServices
```

Status	Name	DisplayName
Running	RPCSS	Remote Procedure Call (RPC)
Running	HTTP	HTTP Service

Figure 4-3

This means that in order for the WinRM service to function, the RPCSS and HTTP services must be running.

We can also look at this from the other end and see what services depend on a specific service.

```
gsv http -DependentServices
```

Status	Name	DisplayName
Stopped	WMSVC	Web Management Service
Running	WinRM	Windows Remote Management (WS-Manag...)
Stopped	Wecsvc	Windows Event Collector
Stopped	w3logsrv	W3C Logging Service
Running	W3SVC	World Wide Web Publishing Service
Stopped	upnphost	UPnP Device Host
Stopped	SSDPSRV	SSDP Discovery
Running	Spooler	Print Spooler
Stopped	RemoteAccess	Routing and Remote Access
Stopped	KPSSVC	KDC Proxy Server service (KPS)
Running	IISADMIN	IIS Admin Service
Stopped	FDResPub	Function Discovery Resource Publica...

Figure 4-4

These are all the services that rely on the HTTP service. Notice WinRM in the list?

## Elevated sessions

You may find that on some platforms, such as Windows 8 or Windows Server 2012, that you can't get detailed information for some services. If that is the case, make sure you have started PowerShell with elevated credentials, even if you are logging on as an admin.

Here's a short script that recursively calls itself to retrieve the required services for a given service.

### Get-Required.ps1

```
#requires -version 2.0

param (
    [Parameter(
        ValueFromPipeline = $false,
        Position = 0,
        Mandatory = $True,
        HelpMessage = "Enter a service name like spooler.")]
    [String]$service,

    [Parameter(
        ValueFromPipeline = $false,
        Position = 1,
        Mandatory = $false,
        HelpMessage= "Enter a computername. The default is the local computer.")]
    [String]$computername = $env:computername
)

#make sure you run this in an elevated PowerShell session

function Get-Required {
#requires -version 2.0
Param([string]$name, [string]$computername)

$errorActionPreference = "SilentlyContinue"

Get-Service -Name $name -Computername $computername -RequiredServices | foreach {
#write the service name to the pipeline
write $_
Get-Required $_.name $computername
#filter out duplicates based on displayname
} | select displayname -Unique | foreach {
#then get the re-get the service object
Get-Service -DisplayName $_.displayname -Computername $computername
}

} #end function

#main script body
try {
    Get-Service -Name $service -Computername $computername -ea stop | Out-Null
    Get-Required $service $computername
}
catch {
    Write-Warning "Failed to find service $service on $computername"
}
Finally {}
```

The script takes a service and computer name as parameters. If you don't specify a computer name, the script defaults to the local computer. The purpose of the script is to display all the services required for a given service to start. For example, the **bits** service requires the **RPCSS** service, which in turn requires **DcomLaunch** and **RpcEptMapper**. The **Get-Required.ps1** script calls the **Get-Required** function, which uses the **Get-Service** cmdlet to find the required services.

```
Get-Service -Name $name -Computername $computername -RequiredServices | foreach {
```

Each required service is then passed to the function again.

```
... | foreach {
#write the service name to the pipeline
write $_

Get-Required $_.name $computername
```

It is likely that some services may have the same requirements, so we need a little sleight of hand to filter out the duplicates.

```
#filter out duplicates based on displayname
} | select displayname -Unique | foreach {

#then get the re-get the service object
Get-Service -DisplayName $_.displayname -Computername $computername
}
```

When executed, you have a list of all services necessary for a given service to run. Here are all the required services for **bits**.

```
c:\scripts\Get-Required.ps1 bits
```

```

Administrator: Windows PowerShell
PS C:\scripts> .\Get-Required.ps1 bits
Status    Name          DisplayName
-----  -----
Running   RpcSS         Remote Procedure Call (RPC)
Running   DcomLaunch    DCOM Server Process Launcher
Running   RpcEptMapper  RPC Endpoint Mapper
Running   EventSystem   COM+ Event System

PS C:\scripts> _

```

Figure 4-5

## Starting services

It should come as no surprise that PowerShell uses the **Start-Service** cmdlet to start a service. You can specify the service by its real name.

```
Start-Service -Name "spooler"
```

Alternatively, you can specify the service by its display name.

```
Start-Service -Displayname "Print spooler"
```

Make sure to use quotes for the display name, since it usually contains spaces. Since the cmdlet doesn't display a result unless there is an error, you can use the **-PassThru** parameter to force the cmdlet to pass objects down the pipeline.

```
Start-Service -Displayname "print spooler" -PassThru
```

Status	Name	DisplayName
-----	-----	-----
Running	Spooler	Print Spooler

## Stopping services

Stopping services is just as easy. Everything we discussed about starting services also applies to stopping services. The only difference is that we use the **Stop-Service** cmdlet.

```
Stop-Service spooler -PassThru
```

```
Administrator: Windows PowerShell
PS C:\> Get-Service -DisplayName "print spooler"
Status    Name            DisplayName
----    --   -----
Stopped  Spooler          Print Spooler

PS C:\> Start-Service -DisplayName "Print spooler"
PS C:\> Stop-Service -DisplayName "Print Spooler" -PassThru
Status    Name            DisplayName
----    --   -----
Stopped  Spooler          Print Spooler

PS C:\>
```

Figure 4-6

## Suspending and resuming services

You can suspend some services. This doesn't stop the service completely—it only pauses it and prevents it from doing anything. Be aware that not all services support suspension.

```
Suspend-Service spooler
```

```
Administrator: Windows PowerShell
PS C:\> Suspend-Service spooler
Suspend-Service : Service 'Print Spooler (spooler)' cannot be suspended because it is not currently running.
At line:1 char:1
+ Suspend-Service spooler
+ ~~~~~
+ CategoryInfo          : CloseError: (System.ServiceProcess.ServiceController:ServiceController) [Suspend-Service], ServiceCommandException
+ FullyQualifiedErrorId : CouldNotSuspendServiceNotRunning,Microsoft.PowerShell.Commands.SuspendServiceCommand

Suspend-Service : Service 'Print Spooler (spooler)' cannot be suspended due to the following error: Cannot pause spooler service on computer '.'.
At line:1 char:1
```

Figure 4-7

You can list the services that have the ability to be paused.

```
Get-Service | Select -Property Name, CanPauseAndContinue
```

If you can suspend a service, you'll see something like the following:

```
Suspend-Service apphostsvc -PassThru
```

```

Administrator: Windows PowerShell
PS C:\> Get-Service apphostsvc
Status      Name            DisplayName
-----      --name--        -----
Running    apphostsvc      Application Host Helper Service

PS C:\> Suspend-Service apphostsvc -PassThru
Status      Name            DisplayName
-----      --name--        -----
Paused     apphostsvc      Application Host Helper Service

PS C:\>

```

Figure 4-8

Use the **Resume-Service** cmdlet to resume a suspended service.

```
Resume-Service apphostsvc -PassThru
```

## Restarting services

If you want to restart a service, you can use the combination of the **Stop-Service** and **Start-Service** cmdlets. However, the simpler approach is to use the **Restart-Service** cmdlet.

```
Restart-Service spooler
```

For services with dependencies, you need to use the **-Force** parameter, otherwise PowerShell will object.

```
Restart-Service lanmanserver
```

This generates an error telling you to use the **-Force** parameter, because it has dependent services. You can see the list of dependent services.

```
Get-Service lanmanserver -DependentServices
```

To restart a service with dependent services, use the **-Force** parameter.

```
Restart-Service lanmanserver -Force
```

```
Administrator: Windows PowerShell
PS C:\> Restart-Service lanmanserver
Restart-Service : Cannot stop service 'Server (Lanmanserver)'
because it has dependent services. It can only be stopped if the
Force flag is set.
At line:1 char:1
+ Restart-Service Lanmanserver
+ ~~~~~
    + CategoryInfo          : InvalidOperation: (System.ServiceProcess.ServiceController:ServiceController) [Restart-Service], ServiceCommandException
    + FullyQualifiedErrorId : ServiceHasDependentServices,Microsoft.PowerShell.Commands.RestartServiceCommand
PS C:\> Restart-Service lanmanserver -Force
WARNING: Waiting for service 'Netlogon (Netlogon)' to start...
PS C:\>
```

Figure 4-9

## Managing services

Use the **Set-Service** cmdlet to change a service's start mode, say from **Automatic** to **Manual**. You can either specify the service by name or display name.

```
Set-Service -Name spooler -StartupType Manual
```

You can change the **-StartupType** parameter to **Automatic**, **Manual**, or **Disabled**. Unfortunately, there are no provisions in the **Get-Service** cmdlet to see the start mode. You'll need to use WMI.

```
Get-WmiObject -class win32_service -filter "name='spooler'"
```

```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject -class win32_service -filter "name='spooler'"

ExitCode : 0
Name : Spooler
ProcessId : 12892
StartMode : Manual
State : Running
Status : OK

PS C:\>
```

Figure 4-10

The **Set-Service** cmdlet also allows you to modify the current status of the service and even allows you to pipe in services (retrieved with the **Get-Service** cmdlet) or service names from a text file.

```
Get-Service | Set-Service -Status "Running"
```

Be careful—that’ll attempt to start every service on your computer! The **Set-Service** cmdlet has several other capabilities, including:

- An **-Include** and **-Exclude** parameter, which allow you to filter the services affected by the cmdlet.
- A **-PassThru** parameter, which outputs whatever services the **Set-Service** cmdlet modifies, so that additional cmdlets can do something with them.
- Parameters for **-Name**, **-DisplayName**, and **-Description**, which allow you to modify the settings of services.
- A **-ComputerName** parameter for managing services on remote computers.

PowerShell has no built-in method for changing the service account or its password. Instead, you’ll have to use WMI—the Win32\_Service class has much broader capabilities—to perform these and additional service-related tasks. If you need to work with services on remote systems, then this is the only way you can accomplish these tasks.

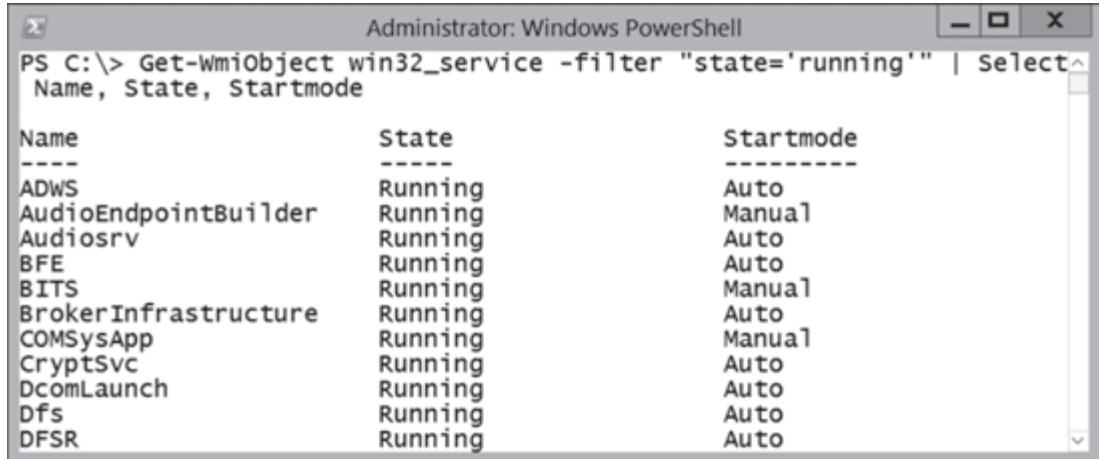
To locate service information with **Get-WmiObject**, use the following basic command.

```
Get-WmiObject win32_service
```

This will provide a list of service information on the local system. There are several ways to filter WMI

information. Here is one approach.

```
Get-WmiObject win32_service -filter "state='running'" | Select Name, State, StartMode
```



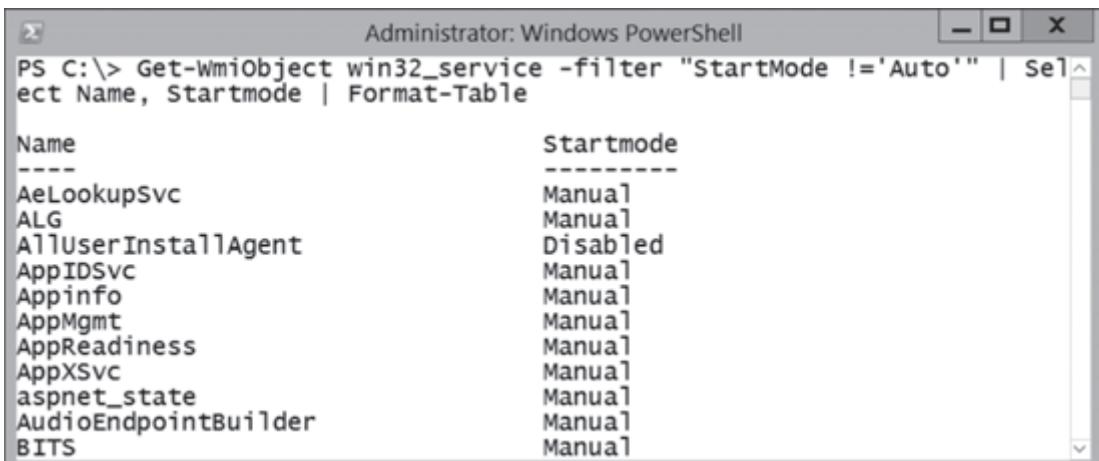
The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command entered is 'PS C:\> Get-WmiObject win32\_service -filter "state='running'" | Select Name, State, Startmode'. The output is a table with three columns: Name, State, and Startmode. The data shows various system services like ADWS, AudioEndpointBuilder, and BITS are running and set to Auto start mode.

Name	State	Startmode
ADWS	Running	Auto
AudioEndpointBuilder	Running	Manual
Audiosrv	Running	Auto
BFE	Running	Auto
BITS	Running	Manual
BrokerInfrastructure	Running	Auto
COMSysApp	Running	Manual
CryptSvc	Running	Auto
DcomLaunch	Running	Auto
Dfs	Running	Auto
DFSR	Running	Auto

Figure 4-11

We've truncated the output to save space. Because we can't do it with the basic service cmdlets, let's find services where the startup type is not **Auto**.

```
Get-WmiObject win32_service -filter "StartMode != 'Auto'" | Select Name, StartMode | Format-Table
```



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command entered is 'PS C:\> Get-WmiObject win32\_service -filter "StartMode !='Auto'" | Select Name, Startmode | Format-Table'. The output is a table with two columns: Name and Startmode. The data shows several services like AeLookupSvc, ALG, and AppIDSvc have their startup type set to Manual.

Name	Startmode
AeLookupSvc	Manual
ALG	Manual
AllUserInstallAgent	Disabled
AppIDSvc	Manual
Appinfo	Manual
AppMgmt	Manual
AppReadiness	Manual
AppXSvc	Manual
aspnet_state	Manual
AudioEndpointBuilder	Manual
BITS	Manual

Figure 4-12

Again, we've truncated the output. One thing to be aware of is that even though we are working with services, there are two different objects. The objects returned by the **Get-Service** cmdlet are **ServiceController** objects and the objects from the **Get-WmiObject** cmdlet are **Win32\_Service** objects. Each object may have a different name for the same property. For example, the **ServiceController** object's property name is "State" while it is "Status" for the **Win32\_Service**. Yet, both will indicate whether a service is running, stopped, or whatever. This doesn't mean that we can't combine the best of both worlds.

```
Get-WmiObject win32_service -Filter "StartMode = 'Disabled'" | Set-Service -Startuptype Manual`  
-WhatIf
```

This snippet takes the output of the **Get-WmiObject** cmdlet, which will return all disabled services, and pipes it to the **Set-Service** cmdlet, which changes the startup type to **Manual**. We've also added the **-WhatIf** parameter, which displays what would have happened.

## Change service logon account

Changing the logon account for a service requires WMI. Suppose you want to change the logon account for the Search service. We'll start by creating an object for the Search service.

```
$svc = gwmi win32_service -filter "name='bits'"
```

We can check the **StartName** property to see the current service account.

```
$svc.StartName
```

To change the service configuration requires us to invoke the **Change()** method. When we read the MSDN documentation for this method <http://tinyurl.com/ltnovf>, we see that the method requires multiple parameters.

```
$svc | Gm -Name Change | fl Definition
```

```

Administrator: Windows PowerShell
PS C:\> $svc=gwmi win32_service -filter "name='bits'"
PS C:\> $svc.StartName
LocalSystem
PS C:\> $svc | Gm -Name Change | fl Definition

Definition : System.Management.ManagementBaseObject
Change(System.String DisplayName, System.String
PathName, System.Byte ServiceType, System.Byte
ErrorControl, System.String StartMode, System.Boolean
DesktopInteract, System.String StartName, System.String
StartPassword, System.String LoadOrderGroup,
System.String[] LoadOrderGroupDependencies,
System.String[] ServiceDependencies)

```

Figure 4-13

Even though we only want to change the **-StartName** and **-StartPassword** parameters, we still have to provide information for the other parameters. In PowerShell, we can pass **\$Null**. This will keep existing settings for the service.

```
$svc.Change($null, $null, $null, $null, $null, $null, "MyCompany\svcUser", "P@ssw0rd")
```

We use **\$Null** for the first six parameters, and then specify the new account and password. We don't have to specify the service parameters after the password, since they aren't changing. They will assumed to be **\$Null**. If you are successful, WMI will return a value of 0 to the screen.

If you receive any other errors, check the MSDN documentation for the error code.

## Controlling services on remote computers

To control services on remote computers, such as starting, stopping, or pausing, you can't use the PowerShell cmdlets. You will have to use the WMI methods. WMI does not have a restart method, but we can achieve the same result by using the WMI service object, **\$svc**.

```
$svc.StopService()
```

```
$svc.StartService()
```

You would use these commands at the end of our change service account process. In fact, there's no reason that you couldn't use them for managing services on the local system as well. If you want to pause or resume a service, the methods are **PauseService()** and **ResumeService()**. In case you are jumping around, make sure to look at the examples above for how to use the **Invoke-WMIMethod** cmdlet.

## Change service logon account password

Changing the service account password uses essentially the same approach as changing the **-StartName** parameter. The only service parameter that is changing is **-StartPassword**.

```
$svc = gwmi win32_service -Filter "name='bits'"  
  
$svc.Change($null, $null, $null, $null, $null, $null, $null, "P@ssw0rd")
```

As we did with changing the service account on a remote computer, you can use the same techniques for changing the service account password as well, by specifying a remote computer name and alternate credentials.

The WMI **Win32\_Service** class isn't compatible with cmdlets like **Stop-Service** and **Start-Service**. That is, you can't take a bunch of **Win32\_Service** objects and pipe them to the **Start-Service** cmdlet—the **-Service** cmdlets only accept service objects generated by the **Get-Service** cmdlet.

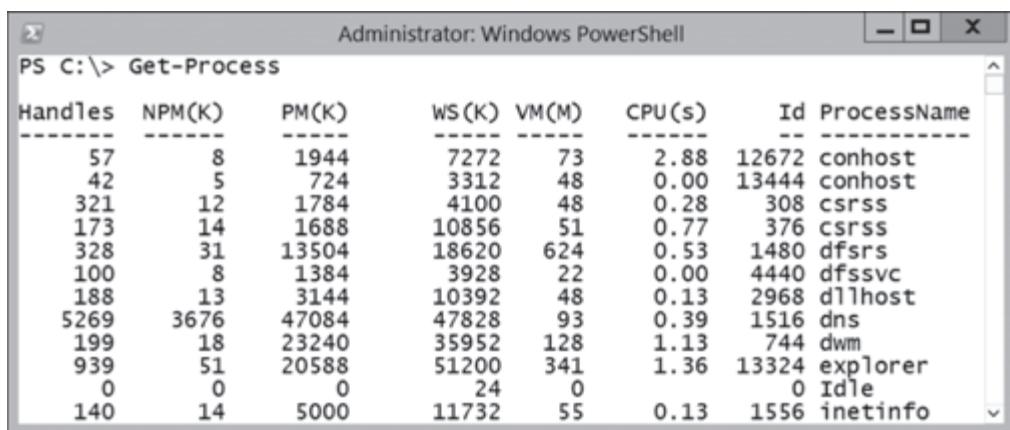
## In Depth 05

# Managing processes

### Managing processes

If you've been with us from the beginning, you're familiar with the **Get-Process** cmdlet. This cmdlet lists the status of all current processes on your system.

Get-Process



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "PS C:\> Get-Process" has been run. The output is a table with the following data:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
57	8	1944	7272	73	2.88	12672	conhost
42	5	724	3312	48	0.00	13444	conhost
321	12	1784	4100	48	0.28	308	csrss
173	14	1688	10856	51	0.77	376	csrss
328	31	13504	18620	624	0.53	1480	dfssrs
100	8	1384	3928	22	0.00	4440	dfssvc
188	13	3144	10392	48	0.13	2968	dllhost
5269	3676	47084	47828	93	0.39	1516	dns
199	18	23240	35952	128	1.13	744	dwm
939	51	20588	51200	341	1.36	13324	explorer
0	0	0	24	0		0	Idle
140	14	5000	11732	55	0.13	1556	inetinfo

Figure 5-1

If you're interested in a specific process, you can reference it by name or by ID.

```
Get-Process notepad
```

The reason you need to know either the process name or ID is so that you can terminate the process with the **Stop-Process** cmdlet, which has an alias of **kill**.

```
notepad
Get-Process -Name notepad
kill 18904
```

```
Administrator: Windows PowerShell
PS C:\> notepad
PS C:\> Get-Process -Name notepad
Handles  NPM(K)      PM(K)      WS(K)    VM(M)      CPU(s)      Id  ProcessName
----  -----      -----      -----    -----      -----      --
87          8          1396       6888     94        0.03    18904  notepad

PS C:\> kill 18904
PS C:\>
```

Figure 5-2

We started Notepad and found the process ID, by using the **Get-Process** cmdlet. Once we identified the process ID, we called the **Stop-Process** cmdlet to terminate it. Yes, there are more efficient ways to accomplish this, but this is just an example to demonstrate the **Stop-Process** cmdlet.

Since the **Get-Process** cmdlet produces objects like all other cmdlets, you can use it in the pipeline.

```
Get-Process | where { $PSItem.handles -gt 1000 } | sort handles -Descending
```

---

**Note:**

You can use `$_` in-place of `$PSItem`—you will see examples both ways.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Get-Process | Where {$PSItem.handles -gt 1000} | Sort handles -Descending
```

The output is a table with the following data:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
5271	3676	47236	47776	94	0.39	1516	dns
1228	115	47112	48380	1169	4.22	480	lsass
1220	50	15832	28796	138	1.72	828	svchost

PS C:\>

Figure 5-3

This expression takes the output of the **Get-Process** cmdlet, filters it with the **Where-Object** cmdlet, and looks for processes with a handle count greater than 1000. PowerShell then pipes the results to the **Sort-Object** cmdlet, which sorts the output by the **Handles** property in descending order.

You can use the **-ComputerName** parameter to retrieve processes from a remote computer. The remote computer does not have to be running PowerShell, but you will need access via RPC and DCOM.

```
Get-Process -Computer "Server"
```

## Detailed process information

When you're working with local processes (as opposed to those from a remote computer), you can retrieve additional useful information. For example, if you'd like to see a list of all modules that each process has loaded, add the **-Module** parameter.

```
Get-Process -Module
```

The resulting list will be very long—you might prefer to limit it to a single process that you're interested in. The **Get-Process** cmdlet has an alias of **ps**.

```
ps notepad -Module | Format-Table -AutoSize
```

```
Administrator: Windows PowerShell
PS C:\> notepad
PS C:\> ps notepad -Module | Format-Table -AutoSize
Size(K) ModuleName      FileName
----- -----
 232 notepad.exe      C:\Windows\system32\notepad.exe
1704 ntdll.dll       C:\Windows\SYSTEM32\ntdll.dll
1248 KERNEL32.DLL    C:\Windows\system32\KERNEL32.DLL
1080 KERNELBASE.dll   C:\Windows\system32\KERNELBASE.dll
 656 ADVAPI32.dll     C:\Windows\system32\ADVAPI32.dll
1296 GDI32.dll        C:\Windows\system32\GDI32.dll
1472 USER32.dll       C:\Windows\system32\USER32.dll
 664 msvcrt.dll      C:\Windows\system32\msvcrt.dll
 616 COMDLG32.dll     C:\Windows\system32\COMDLG32.dll
20240 SHELL32.dll     C:\Windows\system32\SHELL32.dll
 492 WINSPOOL.DRV     C:\Windows\system32\WINSPOOL.DRV
```

Figure 5-4

You can also have the cmdlet display file version information for processes. Again, you can do this for all processes or limit things down to one or more processes that you're interested in. Just use the **-FileVersionInfo** parameter.

```
calc;notepad
ps -Name calc, notepad -FileVersionInfo
```

```
Administrator: Windows PowerShell
PS C:\> calc;notepad
PS C:\> ps -Name calc, notepad -FileVersionInfo
ProductVersion  FileVersion      FileName
----- -----
6.3.9431.0     6.3.9431.0 (w... C:\Windows\system32\calc.exe
6.3.9431.0     6.3.9431.0 (w... C:\Windows\system32\notepad.exe

PS C:\> _
```

Figure 5-5

## Starting a process

On the local system, the easiest and most obvious way to start a process is to simply run a program. You can also run commands as jobs, which will put processes in the background.

```
Start-Process iexplore.exe http://www.sapien.com
```

## Stopping local processes

We can terminate a process with the **Stop-Process** cmdlet by specifying the ID.

```
Stop-Process 1676
```

If we didn't know the process ID, we can also use the following command.

```
Stop-Process -Name notepad
```

Since terminating a process can significantly affect your system, you may want to take advantage of the **-Confirm** parameter to verify you're terminating the correct process.

```
Stop-Process -Id 16060 -Confirm
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `PS C:\> ps -name notepad` is run, displaying a table of process details. Then, the command `PS C:\> Stop-Process -Id 16060 -Confirm` is run, followed by a confirmation dialog:

```

Confirm
Are you sure you want to perform this action?
Performing the operation "Stop-Process" on target "notepad (16060)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend
[?] Help(default is "Y"): _

```

Figure 5-6

We'll explain how to terminate processes on remote servers a little bit later. Note that PowerShell won't normally terminate processes that aren't owned by your user account. If you try to do so, it'll prompt you before actually terminating the processes.

## Waiting on a process

Sometimes, you might want to launch a process, and then have PowerShell—or your script—wait until that process completes before continuing. The **Wait-Process** cmdlet can do just that. Just provide it with an **-ID** or **-Name** parameter, plus either a process ID or name, and it will wait until that process completes before continuing.

---

```

1 Write-Output "Please edit and close Notepad when you are done."
2 Notepad c:\test.ps1
3 Wait-Process -Name Notepad
4 Write-Output "Thank you for closing Notepad."

```

Figure 5-7

You can optionally specify a **-Timeout** parameter (in seconds) to have the process continue after that amount of time, even if the process hasn't ended. If the process does not terminate in the time allotted, then an exception is raised.

```
Wait-Process -Name notepad -Timeout 5
```

## Process tasks

Here are some examples of using PowerShell for common process-management tasks.

### Find Top 10 Processes by CPU Usage

```
Get-process | sort cpu -Descending | select -First 10
```

Here's another example of leveraging the pipeline. The **CPU** property is used to sort the **Get-Process** cmdlet's output in descending order. It then pipes that output to the **Select-Object** cmdlet, which only returns the first 10 items in the list.

### Find Top 10 Processes by Memory Usage

```
Get-Process | sort workingset - Descending | select -First 10
```

The only difference from the previous example is that we are sorting on the **WorkingSet** property.

### Find Top 10 Longest-Running Processes

```
Get-Process | Where { $_.name -ne "system" -and $_.name -ne "Idle" } | sort starttime | ` 
Select -First 10
```

```
Administrator: Windows PowerShell
PS C:\> Get-Process | Where {$_._name -ne "system" -and $_._name -ne "Idle"} | sort starttime | Select -First 10
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
52	2	268	1032	4	0.06	212	smss
321	12	1784	4100	48	0.28	308	csrss
210	15	1700	15020	58	1.06	376	csrss
78	8	716	3812	41	0.09	384	wininit
159	9	1768	7856	71	0.27	412	winlogon
272	11	3292	8512	35	1.05	472	services
1237	116	47032	48368	1169	4.64	480	lsass
342	14	3144	9736	48	0.22	612	svchost
344	17	2528	6636	28	0.14	640	svchost
200	18	23688	39144	131	1.48	744	dwm

Figure 5-8

This example is a little more complex because we want to filter out the **Idle** and **System** processes, by using the **Where-Object** cmdlet.

```
where { $_._name -ne "System" -and $_._name -ne "Idle" }
```

This expression is a compound filter using the **-And** operator to return process objects where the name is not “System” or “Idle.” PowerShell uses the **StartTime** property to sort the remaining process objects. **Select-Object** finally returns the first 10 objects in the list.

How did we know about the **StartTime** property? We used the **Get-Member** cmdlet to see all the possible **Process** object properties.

```
Get-Process | Get-Member
```

## Find process details

Once you know what type of process information you can get, you can execute expressions like the following:

```
Get-Process | Select Name, ID, Company, FileVersion, Path
```

```
Administrator: Windows PowerShell
PS C:\> Get-Process | Select Name, ID, Company, Fileversion, Path

Name      : certsvr
Id        : 1352
Company   : Microsoft Corporation
FileVersion: 6.3.9600.16384 (winblue_rtm.130821-1623)
Path      : C:\Windows\system32\certsrv.exe

Name      : conhost
Id        : 2556
Company   : Microsoft Corporation
FileVersion: 6.3.9600.16384 (winblue_rtm.130821-1623)
Path      : C:\Windows\system32\conhost.exe

Name      : conhost
```

Figure 5-9

In addition to the **Name** and **ID** properties, we've selected the **Company**, **FileVersion**, and **Path** information for each process. Now, you can be better informed about exactly what is running on your server.

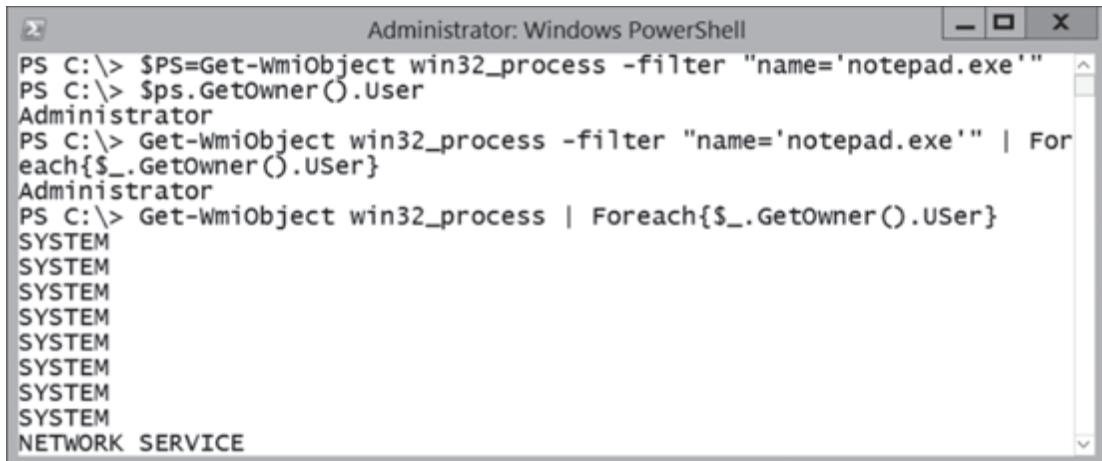
## Find process owners

Another piece of process information that you might find valuable is the process owner. Unfortunately, the **Get-Process** cmdlet doesn't provide access to this information unless you're using PowerShell version 4 and the **-IncludeUserName** parameter. If you are still using an older version, you can use the **Get-WmiObject** cmdlet.

```
$PS = Get-WmiObject win32_process -filter "name='notepad.exe'"
$ps.GetOwner().User
```

This **Get-WmiObject** expression creates a WMI object that has a **GetOwner()** method. The method returns an object with properties of **Domain** and **User**. Once you understand the concept, you can put this together as a one-line expression.

```
Get-WmiObject win32_process -filter "name='notepad.exe'" | Foreach { $_.GetOwner().User }
```



```

Administrator: Windows PowerShell
PS C:\> $PS=Get-WmiObject win32_process -filter "name='notepad.exe'"
PS C:\> $ps.GetUser().User
Administrator
PS C:\> Get-WmiObject win32_process -filter "name='notepad.exe'" | ForEach{$_.GetOwner().User}
Administrator
PS C:\> Get-WmiObject win32_process | Foreach{$_.GetOwner().User}
SYSTEM
SYSTEM
SYSTEM
SYSTEM
SYSTEM
SYSTEM
SYSTEM
SYSTEM
SYSTEM
NETWORK SERVICE

```

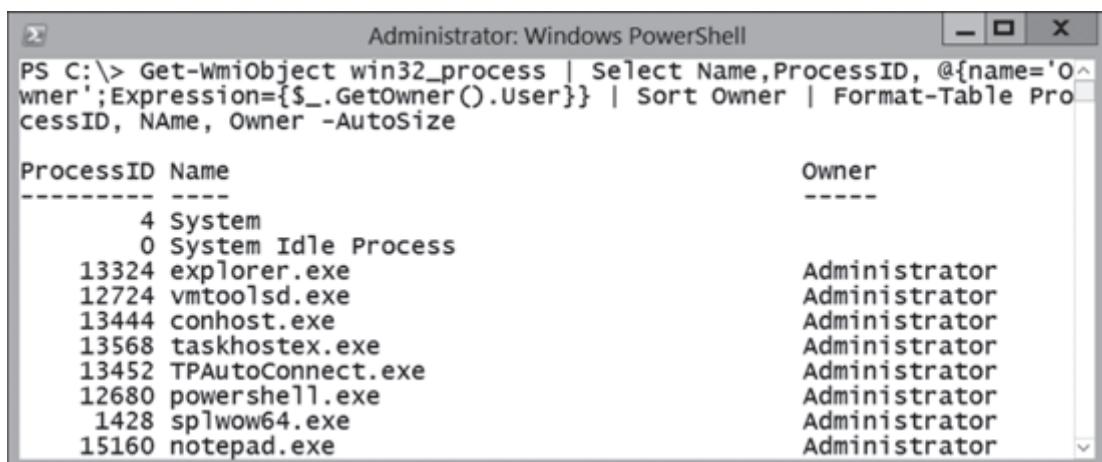
Figure 5-10

We've broken the expression up for printing, but you can type it as one line. Here's an example of how to show the owners of all running processes.

```

Get-WmiObject Win32_Process | ` 
Select ProcessID, Name, @{ Name="Owner" ;Expression = { $_.GetOwner().User } }` 
| Sort owner | Format-Table ProcessID, Name, Owner -AutoSize

```



ProcessID	Name	Owner
4	System	
0	System Idle Process	
13324	explorer.exe	Administrator
12724	vmtoolsd.exe	Administrator
13444	conhost.exe	Administrator
13568	taskhostex.exe	Administrator
13452	TPAutoConnect.exe	Administrator
12680	powershell.exe	Administrator
1428	splwow64.exe	Administrator
15160	notepad.exe	Administrator

Figure 5-11

This example is pretty straightforward until we get to the **Select-Object** cmdlet. In addition to selecting the **ProcessID** and **Name** properties, we're also defining a custom property called **Owner**. The value will be the result of calling the **GetOwner()** method of the current object in the pipeline.

```
Expression = { $_.GetOwner().User }
```

You can pass this property through the pipeline to the **Sort-Object** cmdlet, which in turn sends the output to the **Format-Table** cmdlet for presentation. We'll let you run this on your own to see the results.

## Remote processes

As with most PowerShell cmdlets, most of the process management tools only work on the local system. The **Get-Process** cmdlet is an exception and it returns processes from a remote computer, but offers no way to pass alternate credentials—although you could establish a remote session with alternate credentials, and then use the **Get-Process** cmdlet. Of course, there's nothing wrong with continuing to use WMI to manage processes on remote systems.

```
Get-WmiObject -Class win32_process -Computer DC01 -Credential $cred | select Name, Handle, VirtualSize, WorkingSetSize | Format-Table -AutoSize
```

This example assumes that we've defined the **\$cred** variable earlier by using the **Get-Credential** cmdlet. When using the **Get-WmiObject** cmdlet, you'll receive some additional WMI properties like **\_\_Class**, which generally aren't needed, so use the **Select-Object** cmdlet or choose properties with the **Format-Table** cmdlet to request only the information you want. It is also possible to query for specific processes, by using the **Get-WmiObject** cmdlet.

```
Get-WmiObject -Query "Select * from win32_process where workingsetsize > 10248000" -Computer "DESK61" | Format-Table Name, ProcessID, WorkingSetSize -AutoSize
```

After using the **Get-Process** cmdlet for a while, you may come to expect the same results when querying a remote computer, by using the **Get-WmiObject** cmdlet. You can get the same information with an expression like the following:

```
Get-WmiObject -Query "Select * from win32_process" | `  
Format-Table HandleCount, QuotaNonPagedPoolUsage, PageFileUsage, `  
WorkingSetSize, VirtualSize, KernelModeTime, ProcessID, Name | `  
Format-Table -autosize
```

```

Administrator: Windows PowerShell
PS C:\> Get-WmiObject -Query "Select * from win32_process" |
>> Format-Table HandleCount,QuotaNonPagedPoolUsage,PageFileUsage,
>> WorkingSetSize,VirtualSize,KernelModeTime,ProcessID,Name |
>> Format-Table -autosize
>>

HandleCo QuotaNon PageFile Workings Virtuals KernelMo Process Name
   un PagedPoo Usage etSize  ize deTime ID
   tUsage
-----
    0      0      0     24576  65536 ...56250      0 Syst...
  717      0     112    331776 3436544 ...25000      4 System
    52      3     268   1056768 4317184   625000 212 smss...
   324     12    1784   4202496 50548736 1250000 308 csrs...
  213     16   1700  16523264 61902848 11406250 376 csrs...
    78      8     716   3903488 43118592  937500 384 wini...

```

Figure 5-12

However, if you execute this code, you'll see that the formatting isn't quite what you might expect. This is because PowerShell has defined a default view for the **Get-Process** cmdlet and it handles all the formatting. Here's how you can achieve a similar result with an expression.

```

Get-WmiObject -Query "Select * from win32_process" | sort Name | Format-Table `

@{ Label = "Handles"; Expression = { $_.HandleCount } },
@{ Label = "NPM(K)"; Expression = { "{0:F0}" -f ($_.QuotaNonPagedPoolUsage/1KB) } },
@{ Label = "PM(K)"; Expression = { "{0:F0}" -f ($_.PageFileUsage/1KB) } },
@{ Label = "WS(K)"; Expression = { "{0:F0}" -f ($_.WorkingSetSize/1KB) } },
@{ Label = "VM(M)"; Expression = { "{0:F0}" -f ($_.VirtualSize/1MB) } },
@{ Label = "CPU(s)"; Expression = { "{0:N2}" -f (( $_.KernelModeTime/10000000) + ($_.UserModeTime/10000000)) } },
@{ Label = "ID"; Expression = { $_.ProcessID } },
@{ Label = "ProcessName"; Expression = { $_.Name } } ` -AutoSize

```

```

Administrator: Windows PowerShell
PS C:\> Get-WmiObject -Query "Select * from win32_process" | sort Name
| Format-Table
>> @{$Label="Handles";Expression={$_.HandleCount}},` 
>> @{$Label="NPM(K)";Expression={"{0:F0}" -f ($_.QuotaNonPagedPoolUsage/1KB)}},` 
>> @{$Label="PM(K)";Expression={"{0:F0}" -f ($_.PageFileUsage/1KB)}},` 
>> @{$Label="WS(K)";Expression={"{0:F0}" -f ($_.WorkingSetSize/1KB)}},` 
>> @{$Label="VM(M)";Expression={"{0:F0}" -f ($_.VirtualSize/1MB)}},` 
>> @{$Label="CPU(s)";Expression={"{0:N2}" -f (( $_.KernelModeTime/100000000)+( $_.UserModeTime/10000000))}},` 
>> @{$Label="ID";Expression={$_.ProcessID}},` 
>> @{$Label="ProcessName";Expression={$_.Name}} -AutoSize
>>

Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) ID ProcessName
----- ----- ----- ----- ----- -----
 57    0      2    7212    74   5.70  12672 conhost.exe
 42    0      1    3312    48   0.00  13444 conhost.exe

```

Figure 5-13

PowerShell sorts the results of the **Get-WmiObject** cmdlet, and then sends them to the **Format-Table** cmdlet. We then define custom labels and values by using script blocks, which will give us the same results as the **Get-Process** cmdlet. The added benefit is that we can specify a remote computer. Since this is a lot to type, we recommend creating a script block or a function—and make sure to include a **-ComputerName** parameter so that you can run it on remote computers.

### Get-PS.ps1

```

Function Get-PS {
Param([string]$computer = $env:computername, [System.Management.Automation.PSCredential]$credential)

if ($credential)
{#use alternate credentials if supplied

Get-WmiObject -Query "Select * from win32_process" -Computer $computer ` 
-Credential $credential | sort Name | Format-Table ` 
@{ Label = "Handles"; Expression = { $_.HandleCount } },` 
@{ Label = "NPM(K)"; Expression = { "{0:F0}" -f ($_.QuotaNonPagedPoolUsage/1KB)} },` 
@{ Label = "PM(K)"; Expression = { "{0:F0}" -f ($_.PageFileUsage/1KB)} },` 
@{ Label = "WS(K)"; Expression = { "{0:F0}" -f ($_.WorkingSetSize/1KB)} },` 
@{ Label = "VM(M)"; Expression = { "{0:F0}" -f ($_.VirtualSize/1MB)} },` 
@{ Label = "CPU(s)"; Expression = { "{0:N2}" -f (( $_.KernelModeTime/10000000) +` 
($_.UserModeTime/10000000)) } },` 
@{ Label = "ID"; Expression = { $_.ProcessID } },` 
@{ Label = "ProcessName"; Expression = { $_.Name } } ` 
-AutoSize

}

else
{

Get-WmiObject -Query "Select * from win32_process" -Computer $computer ` 
| sort Name | Format-Table ` 
}

```

```

@{ Label = "Handles"; Expression = { $_.HandleCount } },
@{ Label = "NPM(K)"; Expression = { "{0:F0}" -f ($_.QuotaNonPagedPoolUsage/1KB) } },
@{ Label = "PM(K)"; Expression = { "{0:F0}" -f ($_.PageFileUsage/1KB) } },
@{ Label = "WS(K)"; Expression = { "{0:F0}" -f ($_.WorkingSetSize/1KB) } },
@{ Label = "VM(M)"; Expression = { "{0:F0}" -f ($_.VirtualSize/1MB) } },
@{ Label = "CPU(s)"; Expression = { "{0:N2}" -f (($.KernelModeTime/10000000) +
 ($.UserModeTime/10000000)) } },
@{ Label = "ID"; Expression = { $_.ProcessID } },
@{ Label = "ProcessName"; Expression = { $_.Name } }
-autosize
}
}

```

The following function takes parameters for the computer name and alternate credentials.

```
Get-PS file02 (Get-Credential company\administrator)
```

If a computer name is not passed, the default will be the value of the **%computername%** environment variable.

```
Param([string]$computer = $env:computername,
```

If the user passes an alternate credential object, then PowerShell will call a version of the **Get-WmiObject** script block that uses a **-Credential** parameter.

```
Get-WmiObject -Query "Select * from win32_process" -Computer $computer `
```

```
-Credential $credential | sort Name | Format-Table `
```

Otherwise, the function executes the **Get-WmiObject** cmdlet with the current credentials.

```
Get-WmiObject -Query "Select * from win32_process" -Computer $computer `
```

There are many variations on this you might want to consider. The right approach will depend on your needs.

## Creating a remote process

For a remote machine, you can easily create a remote process by using WMI and the **Invoke-WmiMethod** cmdlet.

```
Invoke-WmiMethod -Comp XP01 -Path win32_process -Name create -ArgumentList notepad.exe
```

We can check whether the command was successful by examining the **ReturnValue** property. Remember that any process you start on a remote system is not necessarily interactive with any logged-on user. This technique is only useful when you want to start some background process on a remote computer that does not require any user intervention. Of course, if you have remote sessions set up with computers, you can simply start a new process by running a command.

## Stopping a remote process

To stop a remote process, you need to use WMI. Begin by obtaining a reference to it, by using the **Get-WmiObject** cmdlet.

```
$calc = Get-Wmiobject -Query "Select * from win32_Process where name='calc.exe'" -Computer FILE02
$calc.Terminate()
```

This expression will terminate the Windows calculator running on FILE02. As with starting processes, the **Terminate()** method will return an object with a **ReturnValue** property. A value of 0 indicates success. You can execute this command as a one-liner as well.

```
(Get-Wmiobject -Query "Select * from win32_Process where name='calc.exe'" -computerFILE02).Terminate()
```

You can use the **Invoke-WmiMethod** cmdlet, but you'll need to know the process ID in order to construct the right path. You should be able to find the process ID, by using either the **Get-Process** cmdlet or the **Get-WmiObject** cmdlet if you need alternate credential support. Once obtained, you can terminate the process.

```
Invoke-WmiMethod -Path "\dc01\root\cimv2:win32_process.handle=3536" -Name terminate
```

This command terminates the Notepad process running on DC01. PowerShell also supports using alternate credentials.

## In Depth 06

# Managing the registry

### Managing the registry

One of the great features in Windows PowerShell is its ability to treat the registry like a file system. Now you can connect to the registry and navigate it just as you would a directory. This is because PowerShell has a registry provider that presents the registry as a drive. That shouldn't come as too much of a surprise, because the registry is a hierarchical storage system much like a file system. So, why not present it as such?

In fact, PowerShell accomplishes this feat for other hierarchical storage types that you experienced earlier in the book. If you run the **Get-PSDrive** cmdlet, you can see the available drives and their providers.

**Get-PSDrive**

Administrator: Windows PowerShell				
Name	Used (GB)	Free (GB)	Provider	Root
A			FileSystem	A:\
Alias			Alias	
C	18.18	41.48	FileSystem	C:\
Cert			Certificate	\
D	3.44		FileSystem	D:\
Env			Environment	
Function			Function	
HKCU			Registry	HKEY_CURRENT_...
HKLM			Registry	HKEY_LOCAL_MA...
Variable			Variable	
WSMan			WSMan	
Z	551.34	147.54	FileSystem	\\vmware-host...

Figure 6-1

You can use the **Set-Location** cmdlet, or its alias **cd**, to change to any of these PSDrives, just as if they were another hard drive in your computer.

```
cd HKLM:\System
dir
```

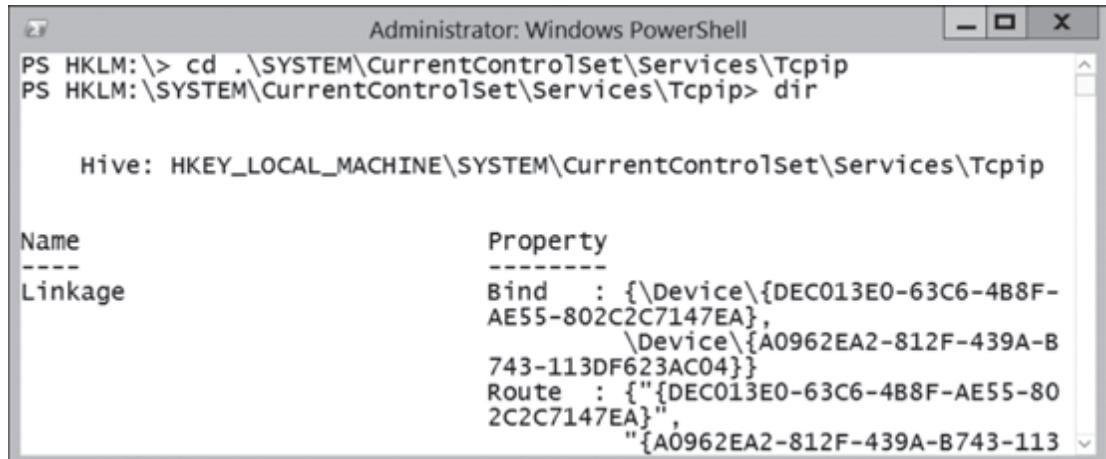
Administrator: Windows PowerShell	
PS C:\>	cd HKLM:\SYSTEM
PS HKLM:\SYSTEM>	dir
	Hive: HKEY_LOCAL_MACHINE\SYSTEM
Name	Property
ControlSet001	Version : 100794368
ControlSet002	Architecture : 9
DriverDatabase	OemInfMap : {255}
HardwareConfig	LastConfig : {efffc4d56-80cc-361a-c866-4f2d6c24bb3e}
	LastId : 0
	BootDriverFlags : 0

Figure 6-2

Remember that the **Tab** key is your friend. Completing long paths—especially in the registry—is easy if you start the folder name and use tab completion to finish it for you. It greatly cuts down on typos and makes finding the path much faster. For TCPIP information in your current control set, do the following:

## Managing the registry

```
cd .\SYSTEM\CurrentControlSet\Services\Tcpip  
dir
```



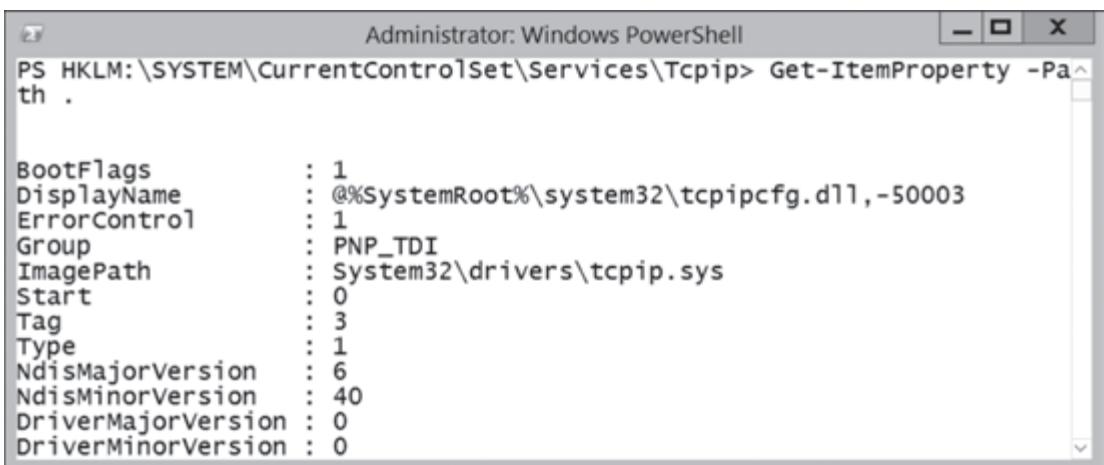
The screenshot shows an Administrator: Windows PowerShell window. The command PS HKLM:\> cd .\SYSTEM\CurrentControlSet\Services\Tcpip was run, followed by PS HKLM:\SYSTEM\CurrentControlSet\Services\Tcpip> dir. The output shows the contents of the registry key HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip. It lists two properties: Bind and Route, each with their respective GUID values.

Name	Property
Linkage	Bind : {\Device\{DEC013E0-63C6-4B8F-AE55-802C2C7147EA}, \Device\{A0962EA2-812F-439A-B743-113DF623AC04}} Route : {"{DEC013E0-63C6-4B8F-AE55-802C2C7147EA}", "{A0962EA2-812F-439A-B743-113DF623AC04}"}

Figure 6-3

For example, if we want to see the keys in our current registry location, we would use an expression, by using **Get-ItemProperty**.

```
Get-ItemProperty -Path .
```



The screenshot shows an Administrator: Windows PowerShell window. The command PS HKLM:\SYSTEM\CurrentControlSet\Services\Tcpip> Get-ItemProperty -Path . was run. The output displays various registry key properties for the Tcpip service, including BootFlags, DisplayName, ErrorControl, Group, ImagePath, Start, Tag, Type, NdisMajorVersion, NdisMinorVersion, DriverMajorVersion, and DriverMinorVersion.

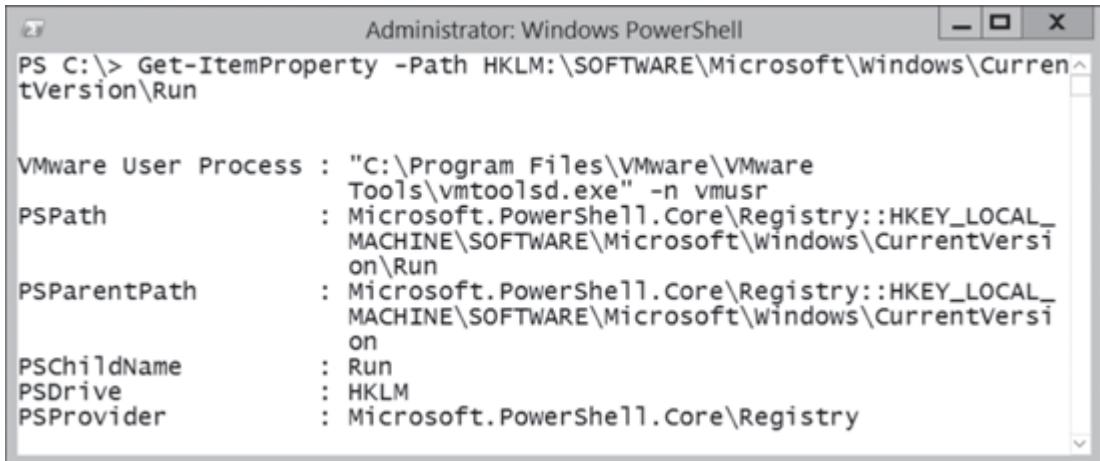
Property	Value
BootFlags	1
DisplayName	@%SystemRoot%\system32\tcpipcfg.dll,-50003
ErrorControl	1
Group	PNP_TDI
ImagePath	System32\drivers\tcpip.sys
Start	0
Tag	3
Type	1
NdisMajorVersion	6
NdisMinorVersion	40
DriverMajorVersion	0
DriverMinorVersion	0

Figure 6-4

In fact, you have to use the **Get-ItemProperty** cmdlet to retrieve any registry keys. You can use this cmdlet without even having to change your location to the registry.

Windows PowerShell: TFM

```
Get-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```



The screenshot shows the Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "Get-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run". The output displays the properties of the registry key:

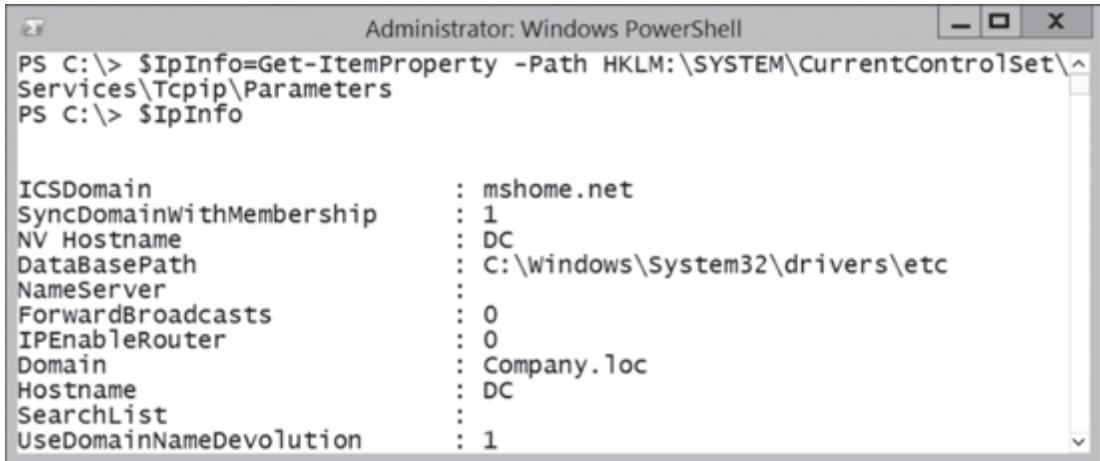
VMware User Process :	"C:\Program Files\VMware\VMware Tools\vmtoolsd.exe" -n vmusr
PSPATH	: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
PSParentPath	: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
PSChildName	: Run
PSDrive	: HKLM
PSProvider	: Microsoft.PowerShell.Core\Registry

Figure 6-5

This expression lists registry keys that indicate what programs are set to run when the computer starts up. We didn't have to change our location to the registry—we only had to specify the registry location as if it were a folder.

You can also create a variable for an item's properties. In Figure 6-6, we find the registry keys for **Parameters**, from our current location, by using the **Get-ItemProperty** cmdlet.

```
$IpInfo = Get-ItemProperty -Path HKLM:\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters  
$IpInfo
```



The screenshot shows the Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "\$IpInfo=Get-ItemProperty -Path HKLM:\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters" followed by "PS C:\> \$IpInfo". The output displays the properties of the registry key:

ICSDomain	:: mshome.net
SyncDomainWithMembership	:: 1
NV Hostname	:: DC
DataBasePath	:: C:\Windows\System32\drivers\etc
NameServer	::
ForwardBroadcasts	:: 0
IPEnableRouter	:: 0
Domain	:: Company.loc
Hostname	:: DC
SearchList	::
UseDomainNameDevolution	:: 1

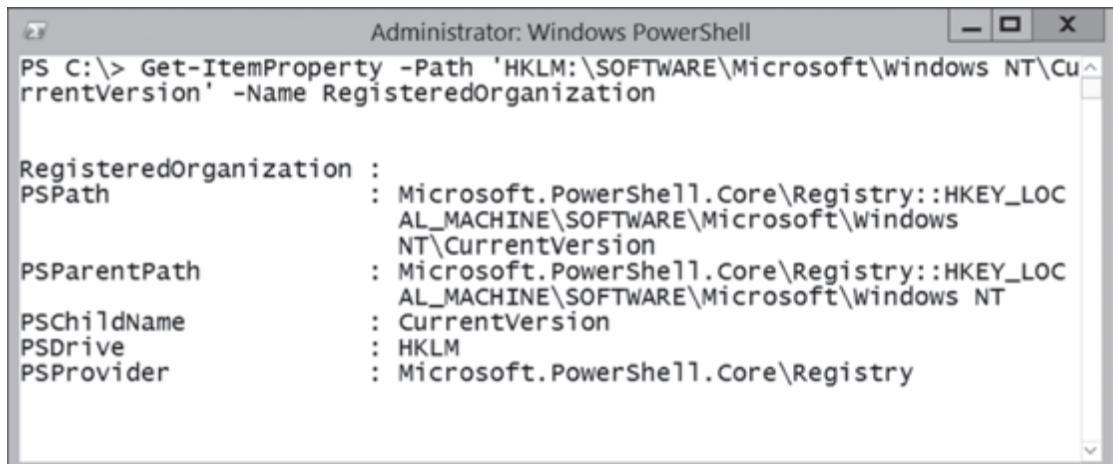
Figure 6-6

By using dot notation with the variable, you can directly find the information you need. We defined the **\$ipInfo** variable to hold the registry keys from HKLM\System\CurrentControlSet\Services\Tcpip\Parameters. Invoking the **\$ipInfo** variable lists all the keys and their values. Alternatively, we can find a specific key and value by using a property name.

```
$IpInfo.dhcpnameserver
$IpInfo.Domain
$IpInfo.HostName
```

We can set a registry value using the **Set-ItemProperty** cmdlet. Say you want to change the **RegisteredOrganization** registry entry. Here's how you find the current value.

```
Get-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion' ` 
-Name RegisteredOrganization
```



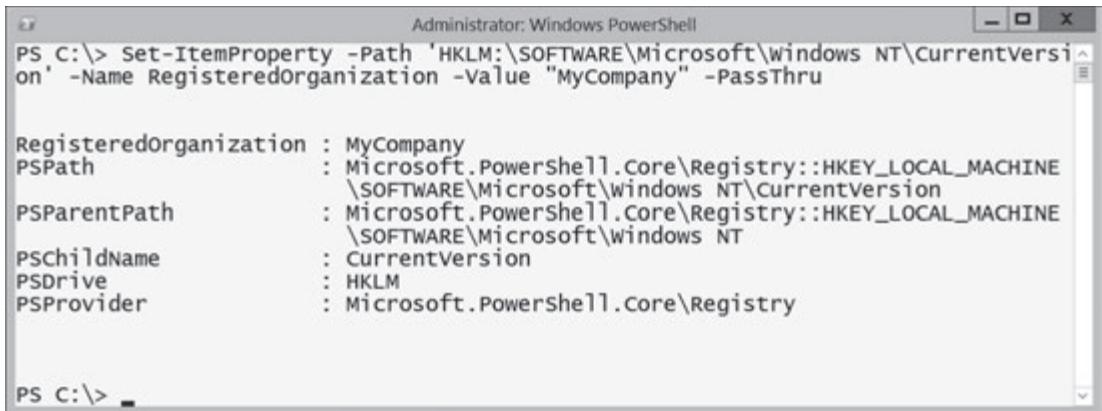
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "Get-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion' -Name RegisteredOrganization". The output displays the properties of the registry entry:

<b>RegisteredOrganization :</b>	
<b>PSPATH</b>	: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
<b>PSParentPath</b>	: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT
<b>PSChildName</b>	: CurrentVersion
<b>PSDrive</b>	: HKLM
<b>PSProvider</b>	: Microsoft.PowerShell.Core\Registry

Figure 6-7

Now let's change it by using the **Set-ItemProperty** cmdlet.

```
Set-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion' ` 
-Name RegisteredOrganization -Value "MyCompany" -PassThru
```



```
Administrator: Windows PowerShell
PS C:\> Set-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion' -Name RegisteredOrganization -Value "MyCompany" -PassThru

RegisteredOrganization : MyCompany
PSPath                 : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE
                           \SOFTWARE\Microsoft\Windows NT\CurrentVersion
PSParentPath            : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE
                           \SOFTWARE\Microsoft\Windows NT
PSChildName             : CurrentVersion
PSDrive                 : HKLM
PSProvider               : Microsoft.PowerShell.Core\Registry

PS C:\>
```

Figure 6-8

By default, the **Set-ItemProperty** cmdlet doesn't write anything to the pipeline, so you wouldn't normally see anything. We've added the **-PassThru** parameter to force the cmdlet to write to the pipeline, so that we can see the result. To properly use the **Set-ItemProperty** cmdlet, you need to specify a path. You can use a period to indicate the current directory or specify a complete path, as in Figure 6-8.

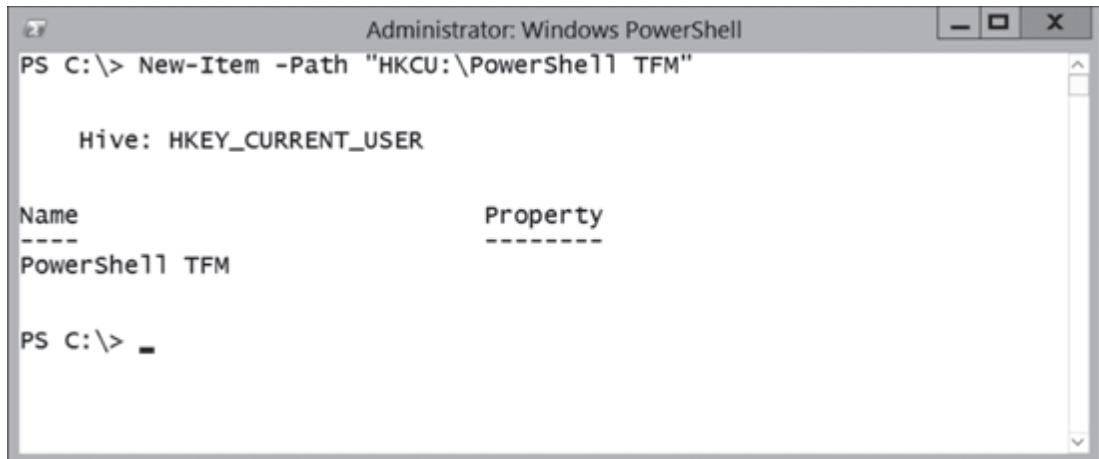
Since accessing the registry in PowerShell is like accessing a file system, you can use recursion, search for specific items, or do a massive search and replace. You can use the **New-Item** and **New-ItemProperty** cmdlets to create new registry keys and properties. Let's change our location to HKEY\_Current\_User and look at the current items in the root.

```
CD HKCU
dir
```

## Creating registry items

In PowerShell, it is very easy to create new registry keys and values. We'll create a new subkey called "PowerShell TFM" under HKCU, by using the **New-Item** cmdlet.

```
New-Item -Path "HKCU:\PowerShell TFM"
```



```

Administrator: Windows PowerShell
PS C:\> New-Item -Path "HKCU:\PowerShell TFM"

Hive: HKEY_CURRENT_USER

Name          Property
----          -----
PowerShell TFM

PS C:\> -

```

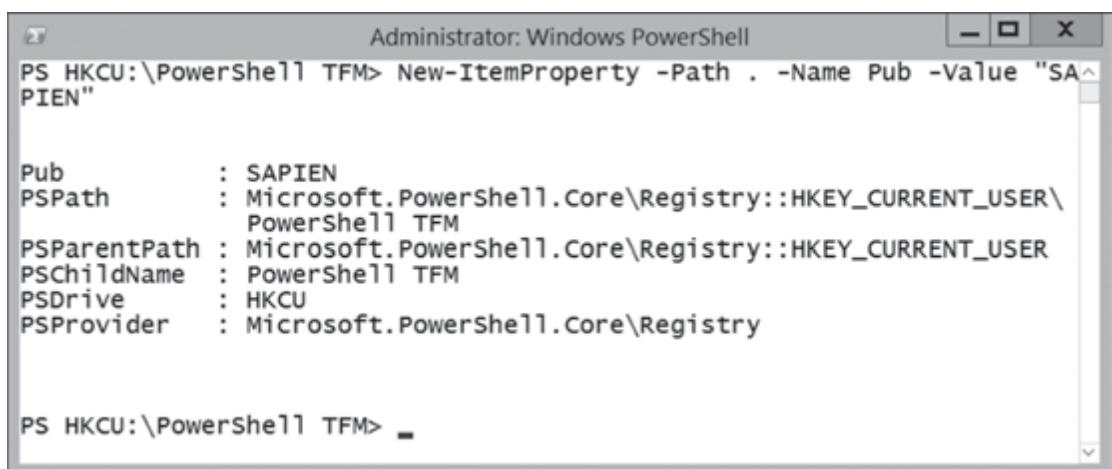
Figure 6-9

You can change to the location to prove that it exists!

```
cd "PowerShell TFM"
```

The **New-Item** cmdlet creates the appropriate type of object because it realizes that we are in the registry. To create registry values, we use the **New-ItemProperty** cmdlet.

```
New-ItemProperty -Path . -Name Pub -Value "SAPIEN"
```



```

Administrator: Windows PowerShell
PS HKCU:\PowerShell TFM> New-ItemProperty -Path . -Name Pub -Value "SAPIEN"

Pub      : SAPIEN
PSPath   : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\
           PowerShell TFM
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName : PowerShell TFM
PSDrive    : HKCU
PSPProvider : Microsoft.PowerShell.Core\Registry

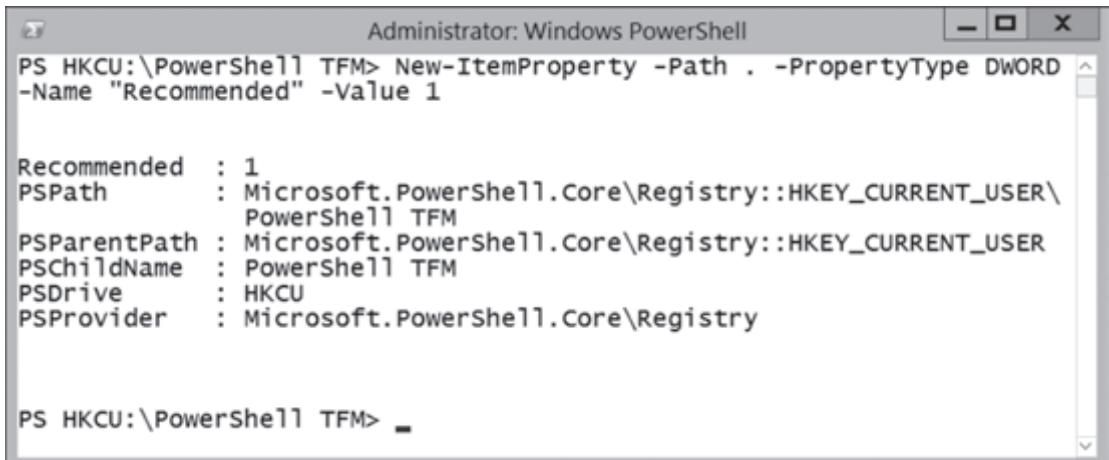
PS HKCU:\PowerShell TFM> -

```

Figure 6-10

We now have a string entry called **Pub**, with a value of SAPIEN. The default property type is **String**. If you want to create a different registry entry, such as a DWORD, use the **-PropertyType** parameter.

```
New-ItemProperty -Path . -PropertyType DWORD -Name "Recommended" -Value 1
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS HKCU:\PowerShell TFM> New-ItemProperty -Path . -PropertyType DWORD -Name "Recommended" -Value 1
```

The output shows the properties of the newly created registry entry:

```
Recommended : 1
PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\
              PowerShell TFM
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName  : PowerShell TFM
PSDrive      : HKCU
PSPrinter    : Microsoft.PowerShell.Core\Registry
```

At the bottom, the prompt is:

```
PS HKCU:\PowerShell TFM> _
```

Figure 6-11

Removing registry items is just as easy, but more dangerous!

## Removing registry items

To remove an item, we call the **Remove-ItemProperty** cmdlet.

```
Remove-ItemProperty -Path . -Name Recommended
```

We use the **Remove-Item** cmdlet to remove the subkey we created.

```
Remove-Item "PowerShell TFM"
```

## Standard registry rules apply

Since PowerShell takes a new approach to managing the registry, take great care in modifying it. Make sure to test your registry editing skills with these new expressions and cmdlets on a test system before even thinking about touching a production server or desktop. Also don't forget to use the **-WhatIf** or **-Confirm** parameters with these cmdlets, to avoid surprises.

## Searching the registry

Searching the registry for information is not that much different from searching any other file system. However, because there is so much information, you'll want to filter the results.

```
dir sapien* -Recurse | Select-Object -Property name
```

```
Administrator: Windows PowerShell
PS HKLM:\SOFTWARE> dir sapien* -Recurse | Select-Object -Property name

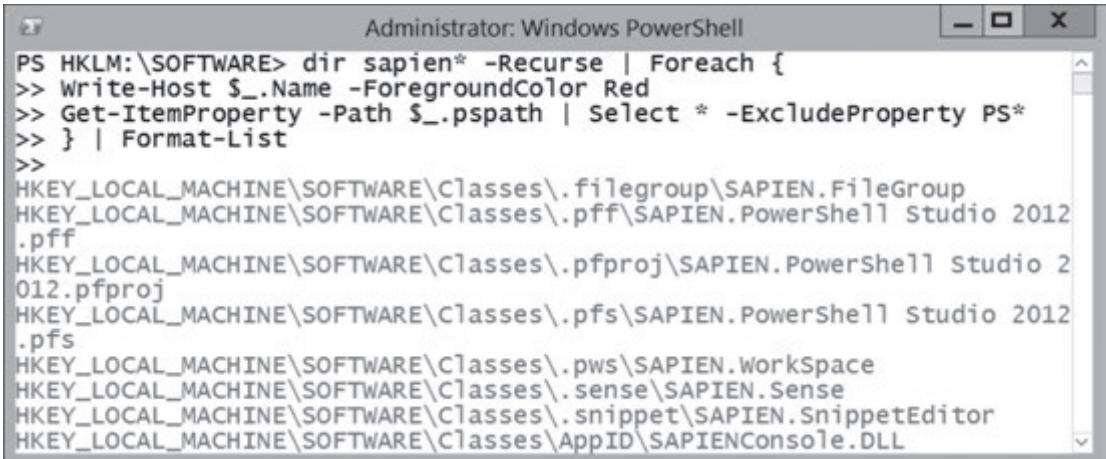
Name
-----
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.filegroup\SAPIEN.FileGroup
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pff\SAPIEN.PowerShell Studio ...
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pfproj\SAPIEN.PowerShell Stud...
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pfs\SAPIEN.PowerShell Studio ...
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pws\SAPIEN.WorkSpace
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.sense\SAPIEN.Sense
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.snippet\SAPIEN.SnippetEditor
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID\SAPIENConsole.DLL
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\SAPIEN.ActiveXPoSH
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\SAPIEN.ActiveXPoSH3
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\SAPIEN.CSVEditor
```

Figure 6-12

From the current location, we're searching for any child keys that match the word "SAPIEN". We took this a step further and enumerated all the matching registry keys.

As you work with the registry in PowerShell, you will realize that you need to use the **Get-ChildItem** cmdlet to enumerate child keys and the **Get-ItemProperty** cmdlet to retrieve values. Sometimes, you need to combine the two cmdlets.

```
dir sapien* -Recurse | foreach {
    Write-Host $_.name -Foregroundcolor Red
    Get-ItemProperty -Path $_.pspath | select * -ExcludeProperty PS*
} | Format-List
```



A screenshot of an Administrator: Windows PowerShell window. The command PS HKLM:\SOFTWARE> dir sapien\* -Recurse | Foreach { is run. The output shows several registry keys under HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes, all starting with 'SAPIEN'. The paths include .filegroup\SAPIEN.FileGroup, .pff\SAPIEN.PowerShell Studio 2012.pff, .pfproj\SAPIEN.PowerShell Studio 2012(pfproj), .pfs\SAPIEN.PowerShell Studio 2012.pfs, .pws\SAPIEN.WorkSpace, .sense\SAPIEN.Sense, .snippet\SAPIEN.SnippetEditor, and \AppID\SAPIENConsole.DLL.

```
PS HKLM:\SOFTWARE> dir sapien* -Recurse | Foreach {
>> Write-Host $_.Name -ForegroundColor Red
>> Get-ItemProperty -Path $_.pspath | Select * -ExcludeProperty PS*
>> } | Format-List
>>
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.filegroup\SAPIEN.FileGroup
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pff\SAPIEN.PowerShell Studio 2012.pff
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pfproj\SAPIEN.PowerShell Studio 2012(pfproj)
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pfs\SAPIEN.PowerShell Studio 2012.pfs
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pws\SAPIEN.WorkSpace
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.sense\SAPIEN.Sense
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.snippet\SAPIEN.SnippetEditor
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID\SAPIENConsole.DLL
```

Figure 6-13

Here's what's going on: from the current directory, we are recursively looking for all keys that start with "SAPIEN".

```
dir sapien* -Recurse | foreach {
```

For each key that we find, PowerShell writes the path to the console, in red.

```
Write-Host $_.name -ForegroundColor Red
```

Then we call the **Get-ItemProperty** cmdlet, by using the **PSPATH** property of the current object. The cmdlet creates additional properties, similar to **PSPATH**, which we want to filter out. So, we pipe to the **Select-Object** cmdlet to exclude these properties.

```
Get-ItemProperty -Path $_.pspath | select * -Exclude PS*
```

At the end, to make the output easier to read, we pipe to the **Format-List** cmdlet, and you can see the results.

Finally, suppose you recall part of a registry value but are not sure of the exact location. We might use a process like in the following example.

```
dir "windows*"-Recurse -ea SilentlyContinue | foreach {
if ( (Get-Itemproperty $_.pspath -Name "RegisteredOwner" -ea SsilentlyContinue).RegisteredOwner )
{
    $_.name
    (Get-ItemProperty $_.pspath -Name "RegisteredOwner").RegisteredOwner
    break
} #end if
} #end foreach
```

We start by recursively searching from the current location. We're using the cmdlet's **-ErrorAction** parameter to turn off the error pipeline. Otherwise, we'll see a lot of errors about non-existing keys. For each item, we'll call the **Get-ItemProperty** cmdlet and check for the **RegisteredOwner** value. If it exists, then we write the name of the registry key and the **RegisteredOwner** value. Finally, we use the **Break** keyword to stop the pipeline and discontinue searching. This is not necessarily a speedy solution, but it appears to be the best we can do with the registry PSDrive provider.

## Managing remote registries with WMI

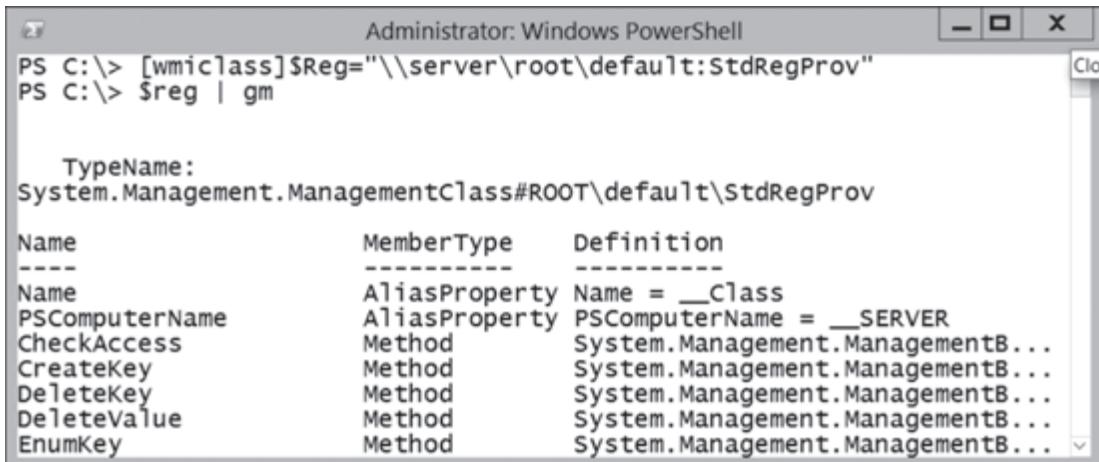
To access remote registries, the easiest approach is to use a remote session and the registry PSDrives. If that is not an option, you can use WMI and the StdReg provider.

```
[WMIClass]$Reg = "root\default:StdRegProv"
```

This will connect you to the local registry via WMI. We recommend you test things locally, first. When you are ready, you can connect to a remote registry by specifying a computer name in the path.

```
[WMIClass]$Reg = "\\\Computername\root\default:StdRegProv"
```

You must be running PowerShell with credentials that have administrative rights on the remote computer. There is no mechanism to specify alternate credentials. Once you have this object, you can use its methods—just pipe the variable to **Get-Member** to see them.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "[wmiclass]\$Reg = '\\server\root\default:StdRegProv'" followed by "\$reg | gm". The output displays the TypeName as "System.Management.ManagementClass#ROOT\default\StdRegProv". A table follows, listing methods and their properties:

Name	MemberType	Definition
Name	AliasProperty	Name = __Class
PSComputerName	AliasProperty	PSComputerName = __SERVER
CheckAccess	Method	System.Management.ManagementB...
CreateKey	Method	System.Management.ManagementB...
DeleteKey	Method	System.Management.ManagementB...
DeleteValue	Method	System.Management.ManagementB...
EnumKey	Method	System.Management.ManagementB...

Figure 6-14

To use the WMI object, almost all the methods will require you to specify a hive constant. Add the following commands to your profile or any remote registry scripts.

```
$HKLM=2147483650
$HKCU=2147483649
$HKCR=2147483648
$KEY_USERS=2147483651
```

Due to the WMI architecture and security, you cannot access the HKEY\_CURRENT\_USER hive on a remote machine. We've included it here for any PowerShell scripts you plan to run locally that will use WMI to access the registry. Most of your remote registry commands will use the constant for HKEY\_LOCAL\_MACHINE.

## Enumerating keys

Use the **EnumKey()** method to enumerate registry keys, starting from a given key.

```
$reg.EnumKey($HKLM, "Software")
```

You need to specify the hive constant and the name of the registry key. In this example, we are enumerating the keys directly in HKEY\_LOCAL\_MACHINE\Software. PowerShell stores the returned values as an array in the **sNames** property.

```
$reg.EnumKey($HKLM, "Software").snames
```

If you wanted to recurse through subkeys, you would need an enumeration function. We'll show you one later.

## Enumerating values

To enumerate values for registry keys, use the **EnumValues()** method.

```
$regpath = "SOFTWARE\Microsoft\Windows\CurrentVersion\Run"
$values = $reg.EnumValues($HKLM, $RegPath)
$values.snames
```

---

**Note:**

you may produce a result if you have nothing in the “Run”.

As with the **EnumKeys()** method, you need to specify a hive and registry path. PowerShell stores the returned values in the **sNames** property, which is why we enumerate them like an array.

In this particular example, we are returning the values of the registry keys in HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run. The semantics Microsoft chose are a little misleading. Even though we're getting values for registry keys, we don't know the data associ-

ated with each key. In the example above, the value may be sufficient. Yet what about something like the following example?

```
$regpath = "SOFTWARE\Microsoft\Windows NT\CurrentVersion"
$values = $reg.EnumValues($HKLM, $RegPath)
$values.snames
```

We need the associated data for each of these key values. The registry provider has several methods for getting key data. However, you need to know what type of data is in each key. A little bit later, we'll show you one way to find the type information.

In the previous example, we know that all the data are strings, so we will use the **GetStringValue()** method.

```
$values.snames | foreach {
    write ("{0} = {1}" -f $_, ($reg.getvalue($HKLM, $regpath, $_).svalue))}
```

The method requires the registry hive, the path, and the value to find. In our example, the value is the **sNames** value coming from the pipeline.

```
"+$reg.GetStringValue($HKLM, $regpath, $_)
```

The code is getting each value and passing it to the **GetStringValue()** method. Here's how you could manually find the value for a single value.

```
$reg.GetStringValue($hkLM, " SOFTWARE\Microsoft\Windows NT\CurrentVersion", "RegisteredOwner")
```

Using WMI to search the registry requires a little fancy footwork. You have to get each key and its values, and then repeat the process for every subkey. We've put together a few functions in the following script to make this easier.

## Get-RegistryPath.ps1

```
Function Get-RegistryPath {
    Param([Management.ManagementObject]$Reg, [int64]$Hive, [string]$regpath)

    #$reg must be previously defined
    #[$WMIClass]$Reg = "root\default:StdRegProv"
    # or
    #[$WMIClass]$Reg = "\servername\root\default:StdRegProv"
    #$Hive is a numeric constant
    # $HKLM=2147483650
    # $HKCU=2147483649
```

```

Function Get-RegistryValue {
    Param([Management.ManagementObject]$Reg,
        [int64]$Hive,
        [string]$regitem,
        [string]$value,
        [int32]$iType)

    #$reg must be previously defined
    #$Reg = [WMIClass]"root\default:StdRegProv"
    # or
    # $Reg = [WMIClass]"\\$servername\root\default:StdRegProv"
    # $Hive is a numeric constant
    # $HKLM = 2147483650
    # $HKCU = 2147483649

    # $regitem is a registry path like "software\microsoft\windows nt\currentversion"
    # $Value is the registry key name like "registered owner"

    # iType is a numeric value that indicates what type of data is stored in the key.
    # Type 1 = String
    # Type 2 = ExpandedString
    # Type 3 = Binary
    # Type 4 = DWord
    # Type 7 = MultiString
    # sample usage:
    # $regPath = "software\microsoft\windows nt\currentversion"
    # $regKey = "RegisteredOwner"
    # Get-RegistryValue $reg $hkLM $regPath $regKey 1

    $obj = New-Object PSObject

    switch ($iType) {
        1 {
            $data = ($reg.GetStringValue($Hive, $regitem, $value)).sValue
        }
        2 {
            $data = ($reg.GetExpandedStringValue($Hive, $regitem, $value)).sValue
        }
        3 {
            $data = "Binary Data"
        }
        4 {
            $data = ($reg.GetDWordValue($Hive, $regitem, $value)).uValue
        }
        7 {
            $data = ($reg.GetMultiStringValue($Hive, $regitem, $value)).sValue
        }
        default {
            $data = "Unable to retrieve value"
        }
    } #end switch

    Add-Member -inputobject $obj -Membertype "NoteProperty" -Name Key -Value $value
    Add-Member -inputobject $obj -Membertype "NoteProperty" -Name KeyValue -Value $data
    Add-Member -inputobject $obj -Membertype "NoteProperty" -Name RegPath -Value $regitem
    Add-Member -inputobject $obj -Membertype "NoteProperty" -Name Hive -Value $hive
    Add-Member -inputobject $obj -Membertype "NoteProperty" -Name KeyType -Value $iType
    write $obj
} #end Get-RegistryValue function

#get values in root of current registry key
$values = $Reg.enumValues($Hive, $regpath)

```

```

if ($values.snames.count -gt 0) {
    for ($i = 0; $i -lt $values.snames.count; $i++) {
        $iType = $values.types[$i]
        $value = $values.snames[$i]
        Get-RegistryValue $Reg $Hive $regpath $value $iType
    }
}

$keys = $Reg.EnumKey($Hive, $regpath)

# enumerate any subkeys
if ($keys.snames.count -gt 0) {
    foreach ($item in $keys.snames) {

        #recursively call this function
        Get-RegistryPath $Reg $hive "$regpath\$item"
    }
}
}

```

The **Get-RegistryPath** function takes parameters for the WMI registry object, the hive constant, and the starting registry path.

```
Get-RegistryPath $reg $HKLM "Software\Microsoft\Windows NT\CurrentVersion"
```

The **Get-RegistryPath** function calls the nested **Get-RegistryValue** function, which returns key data for any keys in the starting location. Then, it enumerates any subkeys. If the count is greater than 0, the function recurses and calls itself. In this way, the function recursively enumerates the entire registry key.

This second function requires a registry provider, hive constant, registry path, key name, and data type as parameters. These are passed from the **Get-RegistryPath** function.

```

$values = $Reg.enumValues($Hive, $regpath)
if ($values.snames.count -gt 0) {
    for ($i = 0; $i -lt $values.snames.count; $i++) {
        $iType = $values.types[$i]
        $value = $values.snames[$i]
        Get-RegistryValue $Reg $Hive $regpath $value $iType
    }
}

```

The **Get-RegistryPath** function checks the collection of values to see if there are any. Assuming there are values, the collection is enumerated. The **\$values** variable actually has two properties. The **sNames** property is every key name and the **Types** property is the corresponding data type. The WMI registry provider doesn't have a good mechanism for discovering what type of data might be in a given key. So, we have to match up the data type with the key name. We then pass these values to the **Get-RegistryValue** function.

This function evaluates the data type by using the **Switch** statement and uses the appropriate method to read the data. It skips binary data and returns a message instead. We then use a custom object to return information. The advantage is that we can use these object properties in formatting the output.

```
Get-RegistryPath $reg $hk1m " SOFTWARE\Microsoft\Windows NT\CurrentVersion" |`  
select Key, KeyValue, RegPath
```

Or you can run a command like the following:

```
Get-RegistryPath $reg $hk1m " SOFTWARE\Microsoft\Windows NT\CurrentVersion" |`  
where {$_.Key -match "registeredowner"}
```

Enumerating or searching the registry with WMI is a slow process. Over the network to a remote computer will be even slower. So if this is your business requirement, don't expect blazing results.

## Modifying the registry

Creating a registry key with the WMI provider is pretty simple.

```
$reg.CreateKey($HKCU, "PowerShellTFM")
```

This will create a key called "PowerShellTFM" under HKEY\_CURRENT\_USER. You can even create a hierarchy with one command.

```
$reg.CreateKey($HKCU, "PowerShellTFM\Key1\Key2")
```

The command will create Key1 and Key2. What about adding values to keys? It depends on the type of data you need to store.

### Create a string value:

```
$reg.SetStringValue($HKCU, "PowerShellTFM\Key1", "SampleKey", "I am a string")
```

### Create a DWORD:

```
$reg.SetDWORDValue($HKCU, "PowerShellTFM\Key1", "Sample Dword", 1024)
```

### Create an expanded string value:

```
$reg.SetExpandedStringValue($HKCU, "PowerShellTFM\Key1\Key2", `  
"Sample Expandable", "%Username%")
```

### Create a multistring value:

```
$reg.SetMultiStringValue($HKCU, "PowerShellTFM\Key1\Key2", `  
"Sample Multi", (Get-Content c:\file.txt))
```

When creating a multistring value, you can't have any blank lines. The WMI method will let you insert blank lines, but when you edit the value with Regedit.exe, it will remove them. Make sure that you have no blank lines to begin with and you should be fine. In all cases, you can check the **Return** value to verify success. A value of 0 indicates the value was successfully written to the registry.

To delete a value, specify the hive, the registry path, and the key name.

```
$reg.DeleteValue($HKCU, "PowerShellTFM\Key1", "SampleKey")
```

To delete a key, specify the hive, and key path.

```
$reg.DeleteKey($HKCU, "PowerShellTFM\Key1\Key2")
```

You delete a key with values, but you can't delete a key with subkeys.

```
$reg.DeleteKey($HKCU, "PowerShellTFM\Key1")
```

If we had used the command above, we would have received a return value of 5, which tells you there are subkeys that you must remove first. In this case, that would be Key2. So, to cleanly delete the keys, we would first need to remove Key2, and then remove Key1.

Working with remote registries via WMI is possible, but it is not the easiest management task you'll face. Write functions and scripts to make it a little easier, but don't expect snappy performance.

## Managing remote registries with the .NET Framework

Another remote registry management alternative is to use the .NET Framework registry classes. You'll find some similarities with the WMI approach. First, we connect to a specific remote registry hive.

```
$computer = "GODOT7"
$regbase = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey("localmachine", $computer)
```

The first parameter for the **OpenRemoteBaseKey()** method is the hive name, which will probably always be HKLM. Use the **OpenSubKey()** method to, well, open a subkey.

```
$key = "software\microsoft\windows nt\currentversion"
$cvs = $regbase.OpenSubKey($key)
```

We always like to pipe new objects to the **Get-Member** cmdlet to discover what they can do.

```
$cvs | gm
```

We can use the **GetValueNames()** method to list all the values and the **GetValue()** method to return the corresponding value.

```
$cv.GetValueNames() | sort | foreach {
    "{0} = {1}" -f $_, $cv.GetValue($_)
}
```

We can also enumerate subkeys.

```
$key = "Software\SAPIEN Technologies, Inc."
$cv = $regbase.OpenSubKey($key)
$cv.GetSubKeyNames()
```

Retrieving registry values is also not especially difficult. You need a subkey object.

```
$k=$regbase.OpenSubKey("Software\RegisteredApplications")
```

Use the **GetValueNames()** method to enumerate it.

```
$k.GetValueNames()
```

Now find the value by using the **GetValue()** method.

```
$k.GetValueNames() | foreach { write ("{0} = {1}" -f $_, $k.GetValue($_)) }
```

The blank entry is from the (DEFAULT) value. The method will work with any data type from String to DWORD to MultiString, and then automatically format the results.

Due to security, creating and modifying registry entries can be a little tricky. Depending on the operating system and credentials, the following may or may not work for you. Everything is pretty much as we've already shown, with the addition of the **Set-Value()** method. The default value is a string, but you can also create other values such as DWORD. If you enter the wrong value type, the error message will give you the correct values, so don't worry too much about making mistakes.

```
$lm=[Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey("localcomputer", $computer)
$lm.CreateSubKey("MYCOMPANY")
$cu.getvaluenames() | foreach { $cu.getValue($_) }
```

As you can see, working with the registry, locally or remotely, takes a bit of work and some actual scripting. We're hoping at some point to have a set of cmdlets that will make this a much easier administrative task.

## In Depth 07

# Regular expressions

### Regular expressions

How often have you tried searching through log files, looking for a particular piece of information or searching for all information that meets a certain format, like an IP address? A regular expression is a text pattern that you can use to compare against a string of text. If the string of text matches the pattern, then you've found what you're looking for. For example, we know that an IP address has the format xxx.xxx.xxx.xxx. We don't know how many integers each octet has, only that there should be four sets of three numbers that are separated by periods. Conceptually, xxx.xxx.xxx.xxx is our pattern and if it's found in the string we are examining, a match is made. Regular expressions can be very complex and confusing at first. We don't have space in this chapter for an exhaustive review of this topic, but we will give you enough information to use basic regular expressions in your PowerShell scripts.

Up to this point, pattern matching has been pretty simple and straightforward. But what if we want to validate that a string was in a particular format, such as a UNC path or an IP address? In this case, we can use regular-expression special characters to validate a string. The following table lists several of the more common special characters.

**Table 7.1 Regular-expression special characters**

<u>Character</u>	<u>Description</u>
\w	Matches a word (alpha-numeric and the underscore character).
\d	Matches any digit (0-9).
\t	Matches any tab.
\s	Matches any whitespace, tab, or newline.

There are additional special characters, but these are the ones you are most likely to use. By the way, each of these characters has an inverse option you can use simply by using the capital letter version. For example, if you want to match a pattern for anything that was not a digit, you would use \D. This is an example of when PowerShell is case sensitive.

PowerShell supports the quantifiers available in .NET regular expressions, as shown in the following table.

**Table 7.2 Regular-expression quantifiers**

<u>Format</u>	<u>Description</u>
Value	Matches exact characters anywhere in the original value.
.	Matches any single character.
[value]	Matches at least one of the characters in the brackets.
[range]	Matches at least one of the characters within the range; the use of a hyphen allows contiguous characters to be specified.
[^]	Matches any character, except those in brackets.
^	Matches the beginning characters.
\$	Matches the end characters.
*	Matches zero or more instances of the preceding character.
?	Matches zero or one instance of the preceding character.
\	Matches the character that follows as an escaped character.
+	Matches repeating instances of the specified pattern, such as abc+.
{n}	Specifies exactly n matches.
{n,}	Specifies at least n matches.
{n,m}	Specifies at least n, but no more than m, matches.

By combining pattern matching characters with these qualifiers, it is possible to construct complex regular expressions.

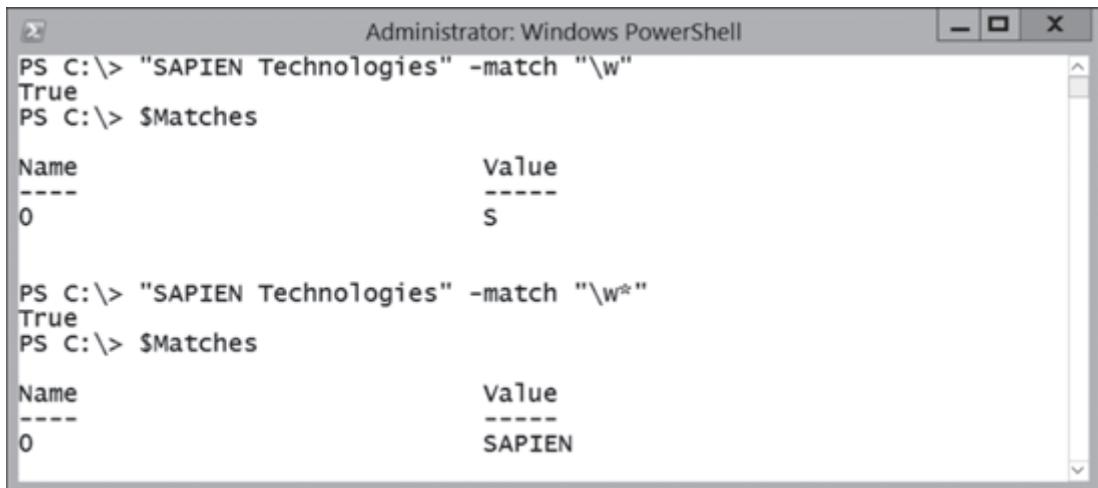
## Writing regular expressions

The built-in variable **\$Matches** will show what values matched when PowerShell examined a regular expression. The following are some simple, regular-expression examples.

```
"SAPIEN Technologies" -match "\w"
$Matches
```

While this example produces a **True** result, notice the match was for the 'S' in SAPIEN. The example in Figure 7-1 matches the first word.

```
"SAPIEN Technologies" -match "\w*"
$Matches
```



```
Administrator: Windows PowerShell
PS C:\> "SAPIEN Technologies" -match "\w"
True
PS C:\> $Matches
Name          Value
----          -----
0              S

PS C:\> "SAPIEN Technologies" -match "\w*"
True
PS C:\> $Matches
Name          Value
----          -----
0              SAPIEN
```

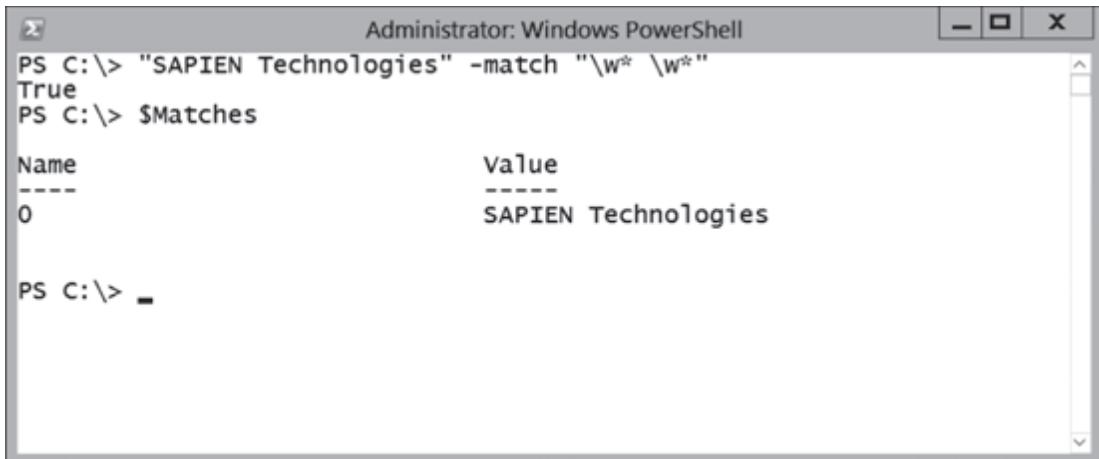
Figure 7-1

The following example also produces the match for the first word.

```
"SAPIEN Technologies" -match "\w+"
$Matches
```

Here is another example, where the entire phrase is matched.

```
"SAPIEN Technologies" -match "\w* \w*"
$Matches
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> "SAPIEN Technologies" -match "\w\* \w\*"". The output shows "True" and then the variable \$Matches is displayed as a table:

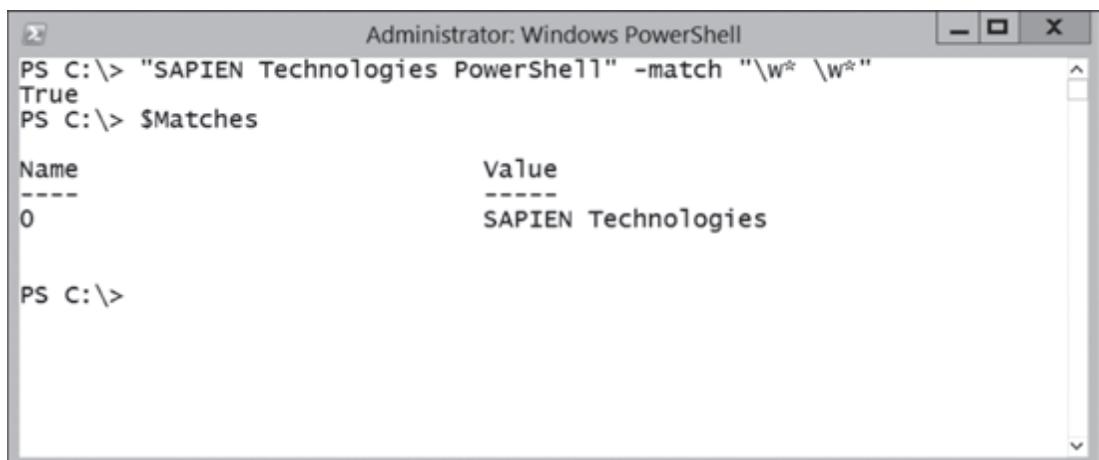
Name	Value
0	SAPIEN Technologies

Figure 7-2

So, let's take a look at what's going on here. The first example compares the string "SAPIEN Technologies" to the pattern `\w`. Recall that comparison results are automatically stored in the `$Matches` variable. As you can see, `\w` matches "S". Why doesn't it match "SAPIEN" or "SAPIEN Technologies"? The `\w` pattern means any word, even a single-character word. If we want to match a complete word, then we need to use one of the regular-expression qualifiers. For example, you can see the second and third examples use `\w*` and `\w+` respectively. In this particular example, these patterns return the same results.

If we want to match a two word phrase, then we would use an example like the last one using `\w* \w*`. If we were testing a match for "SAPIEN Technologies PowerShell", then we'd use something like the example in Figure 7-3.

```
"SAPIEN Technologies PowerShell" -match "\w* \w*"
$Matches
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> "SAPIEN Technologies PowerShell" -match "\w\* \w\*"". The output shows "True" and then the variable \$Matches is displayed as a table:

Name	Value
0	SAPIEN Technologies

Figure 7-3

This also matches, but as you can see it only matches the first two words. If we want to specifically match a two-word pattern, then we need to qualify our regular expression so that it starts and ends with a word.

```
"SAPIEN Technologies PowerShell" -match "^\\w* \\w*$"  
"SAPIEN Technologies" -match "^\\w* \\w*$"
```

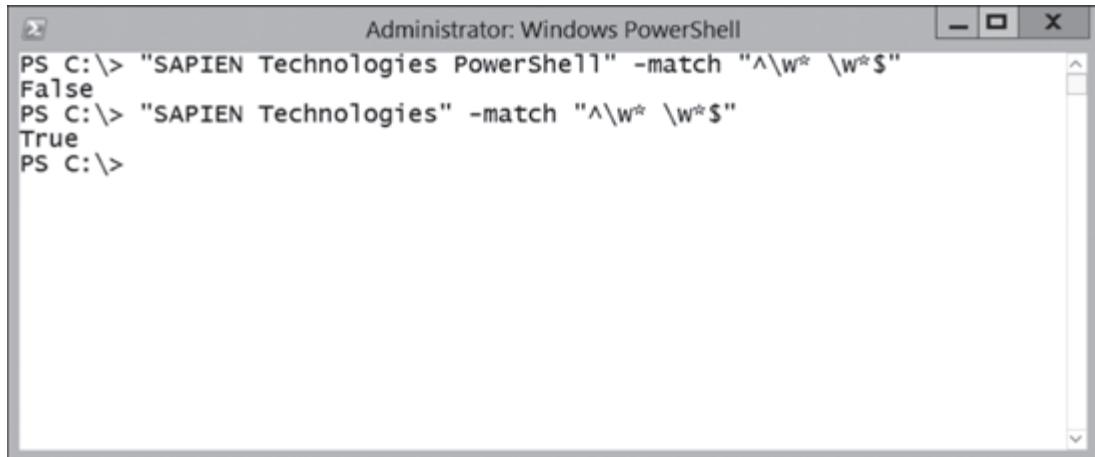
A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The window shows the command "PS C:\> "SAPIEN Technologies PowerShell" -match "^\\w\* \\w\*\$" followed by the output "False". Below it, the command "PS C:\> "SAPIEN Technologies" -match "^\\w\* \\w\*\$" is shown with the output "True". The window has standard operating system window controls (minimize, maximize, close) at the top right.

Figure 7-4

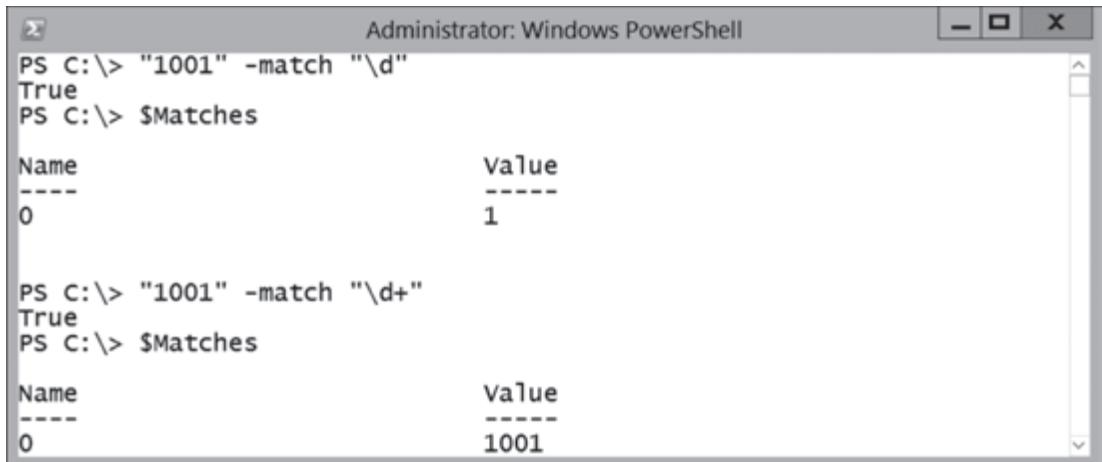
The recommended best practice for strict, regular-expression evaluation is to use the ^ and \$ qualifiers to denote the beginning and end of the matching pattern.

Here's another example.

```
"1001" -match "\\d"  
$Matches
```

```
"1001" -match "\\d+"
$Matches
```



```

Administrator: Windows PowerShell

PS C:\> "1001" -match "\d"
True
PS C:\> $Matches

Name          Value
----          -----
0             1

PS C:\> "1001" -match "\d+"
True
PS C:\> $Matches

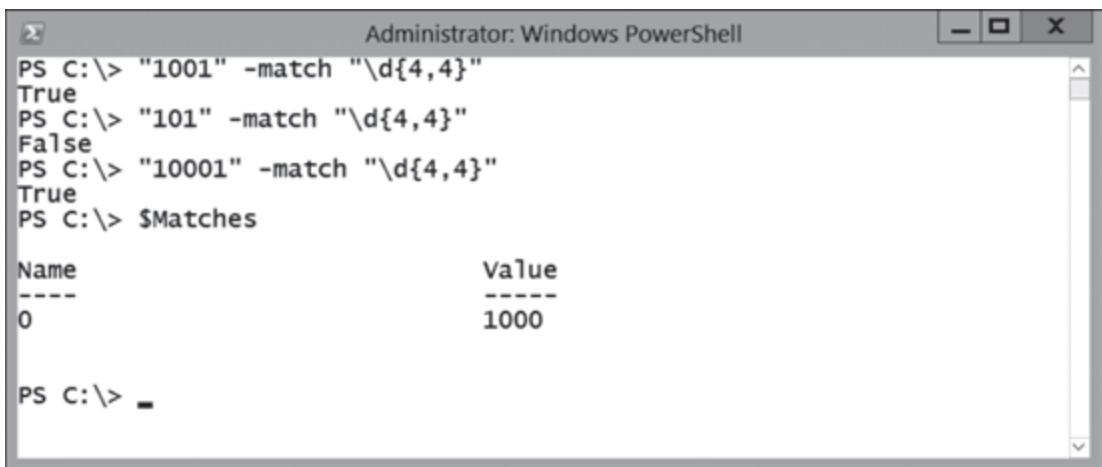
Name          Value
----          -----
0             1001
  
```

Figure 7-5

In the first example, we used the digit-matching pattern and received a **True** result. However, **\$Matches** shows it only matched the first digit. Using **\d+** in the second example returns the full value. If we want the number to be four digits, then we can use a qualifier.

```

"1001" -match "\d{4,4}"
"101" -match "\d{4,4}"
"10001" -match "\d{4,4}"
  
```



```

Administrator: Windows PowerShell

PS C:\> "1001" -match "\d{4,4}"
True
PS C:\> "101" -match "\d{4,4}"
False
PS C:\> "10001" -match "\d{4,4}"
True
PS C:\> $Matches

Name          Value
----          -----
0             1000

PS C:\>
  
```

Figure 7-6

The qualifier **{4,4}** indicates that we want to find a string with at least four matches. In this case, that would be an integer (**\d**) and no more than 4. When we use the regular expression to evaluate 101, it returns **False**. Notice that “10001” tests **True**, because it has four digits, but the match was only on the first 4.

## UNC

Figure 7-7 shows a regular expression that is evaluating a simple UNC path string.

```
"\\server\share" -match "^\\\\\\w*\\\\w*$"
$Matches
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> "\\Server\Share" -match "^\\\\\\w*\\\\w*$"
True
PS C:\> $Matches
```

A table is displayed showing the results of the match:

Name	Value
0	\\Server\Share

PS C:\>

Figure 7-7

This example looks a little confusing, so let's break it apart. First, we want an exact match, so we're using ^ and \$ to denote the beginning and end of the regular expression. We know the server name and path will be alphanumeric words, so we can use \w. Since we want the entire word, we'll use the \* qualifier. All that's left are the \\ and \ characters in the UNC path. Remember that \ is a special character in regular expressions and that if we want to match the \ character itself, then we need to escape it by using another \ character. In other words, each \ will become \\. So, the elements of the regular expression break down to:

1. ^ (beginning of expression)
2. \\ becomes \\\\
3. \w\* (servername)
4. \ becomes \\
5. \w\* (sharename)
6. \$ (end of expression)

Putting this all together, we end up with ^\\\\\\w\*\\\\w\*\$\$. As you can see in the example, this is exactly what we get.

Notice that **\$Matches** indicates the match at index 0, which is fine—assuming we want a complete match. However, we can also use regular expressions to match individual components by grouping each pattern.

### Windows PowerShell: TFM

```
"\\server01\public" -match "(\\\\\\w*)\\\\(\w*)"  
$Matches
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is `"\\server01\public" -match "(\\\\\\w*)\\\\(\w*)"`. The output shows the matches indexed from 0 to 2. A table is displayed:

Name	Value
2	public
1	\\server01
0	\\server01\public

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

Figure 7-8

You're probably thinking, "So what?" Well, regular expressions in PowerShell support a feature called Captures. This allows us to define a name for the capture instead of relying on the index number. The format is to use "`?<capturename>`" inside parentheses of each pattern.

```
"\\server01\public" -match "(?<server>\\\\\\w*)\\\\(?<share>\\w*)"  
$Matches
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is `"\\server01\public" -match "(?<server>\\\\\\w*)\\\\(?<share>\\w*)"`. The output shows the matches named "server" and "share". A table is displayed:

Name	Value
server	\\server01
share	public
0	\\server01\public

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

Figure 7-9

We still have the complete match at index 0, but notice that there are now names associated with the other matches. Now we can reference these elements directly by name.

```
$Matches.server
$Matches.share
```

Using named captures makes it much easier to work with the **\$Matches** variable.

## IP addresses

Let's look at another example. The following is a regular-expression pattern to match an IP address.

```
"192.168.100.2" -match "^\\d+\\.\\d+\\.\\d+\\.\\d+$"
$Matches
```

```
"192.168.100" -match "^\\d+\\.\\d+\\.\\d+\\.\\d+$"
$Matches
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It contains the following command history and output:

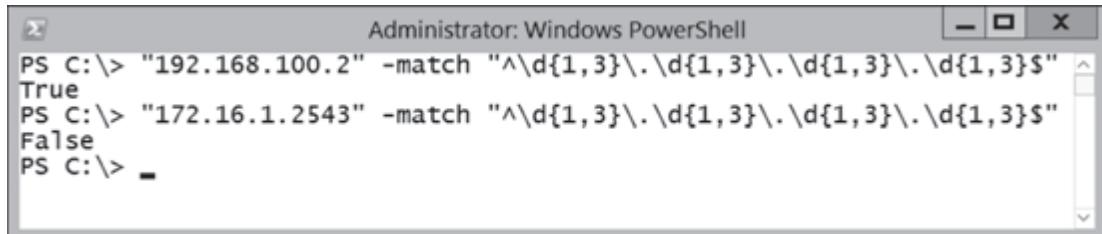
```
PS C:\> "192.168.100.2" -match "^\\d+\\.\\d+\\.\\d+\\.\\d+$"
True
PS C:\> $Matches
Name          Value
----          -----
0            192.168.100.2

PS C:\> "192.168.100" -match "^\\d+\\.\\d+\\.\\d+\\.\\d+$"
False
PS C:\> $Matches
Name          Value
----          -----
0            192.168.100.2
```

Figure 7-10

This should begin to look familiar. We're matching digits and using the \ character to escape the period character, since the period is a regular-expression special character. By using the beginning and end of regular-expression characters, we also know that we'll only get a successful match on a string with four numbers that are separated by periods. Of course, there is more to an IP address than four numbers. Each set of numbers can't be greater than three digits long. Figure 7-11 shows how we can construct a regular expression to validate that.

```
Windows PowerShell: TFM  
"192.168.100.2" -match "^\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}$"  
"172.16.1.2543" -match "^\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}$"
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It contains the following command and output:

```
PS C:\> "192.168.100.2" -match "^\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}$"  
True  
PS C:\> "172.16.1.2543" -match "^\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}$"  
False  
PS C:\> -
```

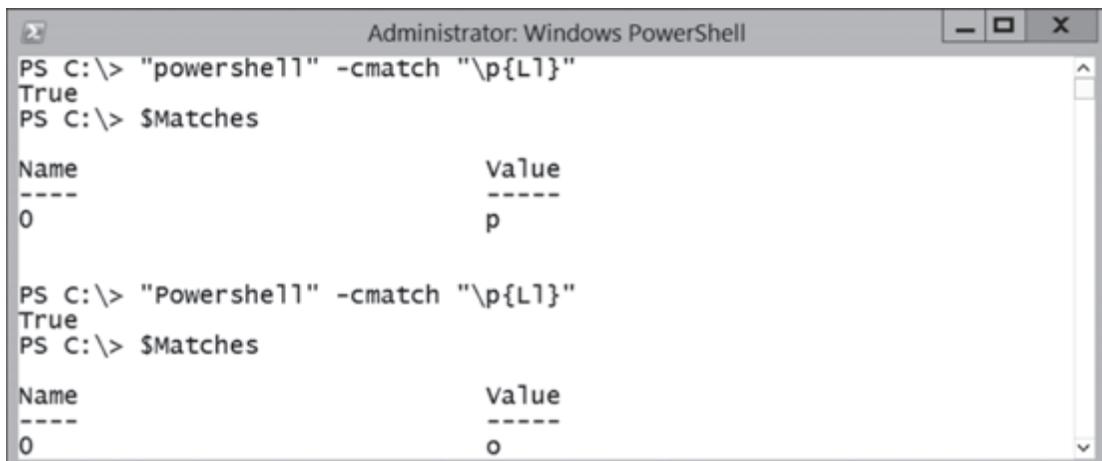
Figure 7-11

The first example in Figure 7-11 matches because each dotted octet is between one and three digits. The second example fails because the last octet is four digits.

## Named character sets

PowerShell regular expressions also support named character sets, such as in the example in Figure 7-12.

```
"powershell" -cmatch "\p{L1}"  
"Powershell" -cmatch "\p{L1}"
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It contains the following command and output:

```
PS C:\> "powershell" -cmatch "\p{L1}"  
True  
PS C:\> $Matches  


| Name | Value |
|------|-------|
| 0    | p     |

  
PS C:\> "Powershell" -cmatch "\p{L1}"  
True  
PS C:\> $Matches  


| Name | Value |
|------|-------|
| 0    | o     |


```

Figure 7-12

The named-character set syntax is to use `\p` and the set name, in this case `{L}` to indicate to only match on lowercase letters. We also told PowerShell to check for case by using `-cmatch`. This is functionally the same as the following example.

```
"powershell" -cmatch "[a-z]"
```

This may not seem like much of an improvement, but using character classes can simplify your regular expressions.

```
"PowerShell" -cmatch "\p{L}+"
```

The `{L}` character class indicates any uppercase or lowercase character. We could receive the same result with the example in Figure 7-13.

```
"PowerShell" -match "[a-zA-Z]+"
```

```
Administrator: Windows PowerShell
PS C:\> "PowerShell" -cmatch "\p{L}+"
True
PS C:\> $Matches
Name          Value
----          -----
0            PowerShell

PS C:\> "PowerShell" -cmatch "[a-zA-Z]@"
True
PS C:\> $Matches
Name          Value
----          -----
0            PowerShell
```

Figure 7-13

The character set requires a little less typing. As your expressions grow in length and complexity, you will appreciate this.

You can use any of the Unicode character sets in the following table.

**Table 7.3 Regular-expression, Unicode character sets**

<u>Character Set</u>	<u>Description</u>
Cc	Other, Control
Cf	Other, Format
Cn	Other, Not Assigned (no characters have this property)
Co	Other, Private Use
Cs	Other, Surrogate
Ll	Letter, Lowercase
Lm	Letter, Modifier
Lo	Letter, Other
Lt	Letter, Title Case (e.g., Microsoft Windows)
Lu	Letter, Uppercase
Mc	Mark, Spacing Combining
Me	Mark, Enclosing
Mn	Mark, Non spacing
Nd	Number, Decimal Digit
Nl	Number, Letter
No	Number, Other
Pc	Punctuation, Connector
Pd	Punctuation, Dash
Pe	Punctuation, Close
Pf	Punctuation, Final Quote
Pi	Punctuation, Initial Quote
Po	Punctuation, Other
Ps	Punctuation, Open
Sc	Symbol, Currency
Sk	Symbol, Modifier
Sm	Symbol, Math
So	Symbol, Other
Zl	Separator, Line
Zp	Separator, Paragraph
Zs	Separator, Space

The .NET Framework also provides other grouping categories for the character sets shown above.

**Table 7.4 Additional regular-expression groupings**

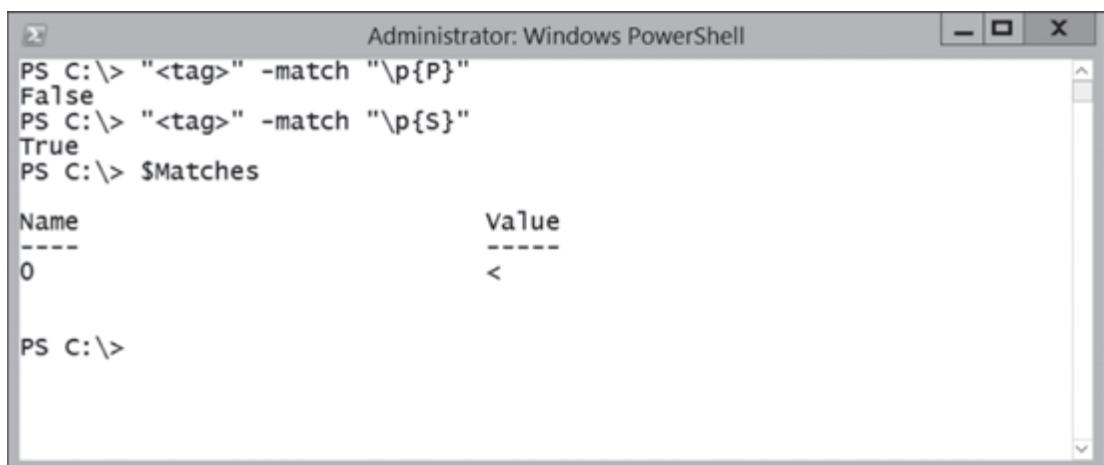
<u>Character Set</u>	<u>Grouping</u>
C	All control characters Cc, Cf, Cs, Co, and Cn.
L	All letters Lu, Ll, Lt, Lm, and Lo.
M	All diacritical marks Mn, Mc, and Me.
N	All numbers Nd, Nl, and No.
P	All punctuation Pc, Pd, Ps, Pe, Pi, Pf, and Po.
S	All symbols Sm, Sc, Sk, and So.
Z	All separators Zs, Zl, and Zp.

You might have to experiment with these sets because they may not all be self-evident, as shown in the following example.

```
"<tag>" -match "\p{P}"
```

The previous example will return **FALSE** because < is considered a symbol, not punctuation.

```
"<tag>" -match "\p{S}"
$Matches
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "<tag>" -match "\p{P}" returns \$False. The command "<tag>" -match "\p{S}" returns \$True. The command \$Matches displays a table with one row, where the Name column is 0 and the Value column is '<'.

Name	Value
0	<

Figure 7-14

## Select-String

PowerShell includes a pattern-matching cmdlet called **Select-String**. The intent is that you'll be able to select strings from text files that match the pattern. The pattern can be as simple as "ABC" or a regular expression like "\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}". This cmdlet is like the **grep** and **findstr** commands found in other shells.

For example, you might have an audit file of user activity and you want to find all lines that include the user account for Tybald Rouble.

```
Get-Content audit.txt | Select-String -Pattern "mydomain\trouble"
```

PowerShell will display every line with the pattern "mydomain\trouble".

When used with the **Get-ChildItem** cmdlet, you can quickly search an entire directory of files for specific strings. This is especially useful when searching multiple log files such as with Microsoft IIS.

Here is an example of searching Windows logs with the **Get-Eventlog** cmdlet.

```
Get-EventLog -LogName System | Select message, timewritten |
Select-String -Pattern "Windows Installer"
```

```
Administrator: Windows PowerShell
PS C:\> Get-EventLog -LogName System | Select message, timewritten | Select-String -Pattern "Windows Installer"
The Windows Installer service e... 7/6/2013 2:59:43 PM
The Windows Installer service e... 7/6/2013 2:53:37 PM
The Windows Installer service e... 7/1/2013 9:27:14 AM
The Windows Installer service e... 7/1/2013 9:21:38 AM
The Windows Installer service e... 6/29/2013 10:53:29 AM
The Windows Installer service e... 6/29/2013 10:30:24 AM
The Windows Installer service e... 6/26/2013 5:31:12 PM
The Windows Installer service e... 6/26/2013 5:26:11 PM
The Windows Installer service e... 6/26/2013 5:20:14 PM
The Windows Installer service e... 6/26/2013 5:10:48 PM

PS C:\>
```

Figure 7-15

The example in Figure 7-15 will search the local event log for events and display those with the pattern "Windows Installer" in the message.

The **Select-String** cmdlet can also use regular-expression patterns.

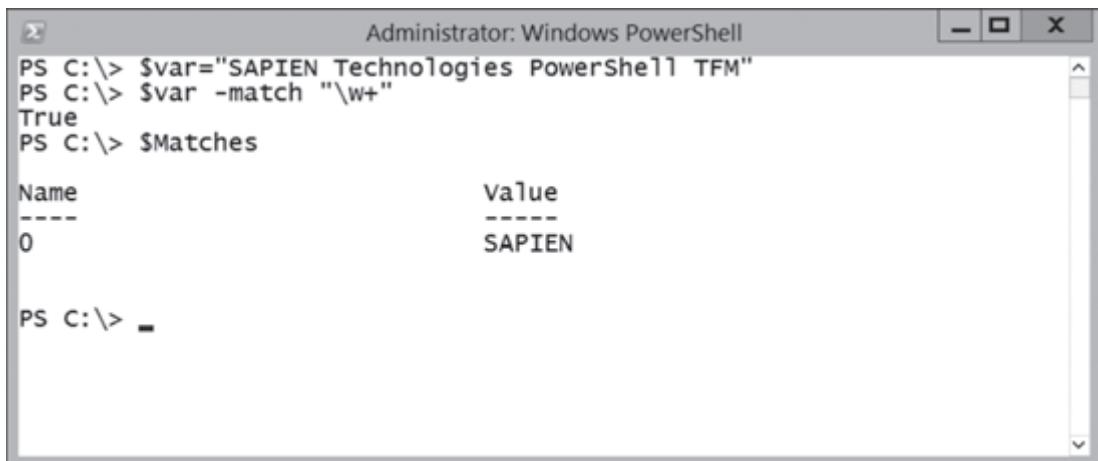
```
cat c:\iplist.txt | Select-String "(172.16.)\d{2,3}\.\d{1,3}"
```

This regular-expression will select all strings from IPLList.txt that start with 172.16 and where the third octet has either two or three digits. This pattern will match on strings like 172.16.20.124, but not on 172.16.2.124.

## Regex object

When you use the **-Match** operator, even with a regular-expression pattern, the operation only returns the first match found.

```
$var = "Sapien Technologies PowerShell TFM"
$var -match "\w+"
```



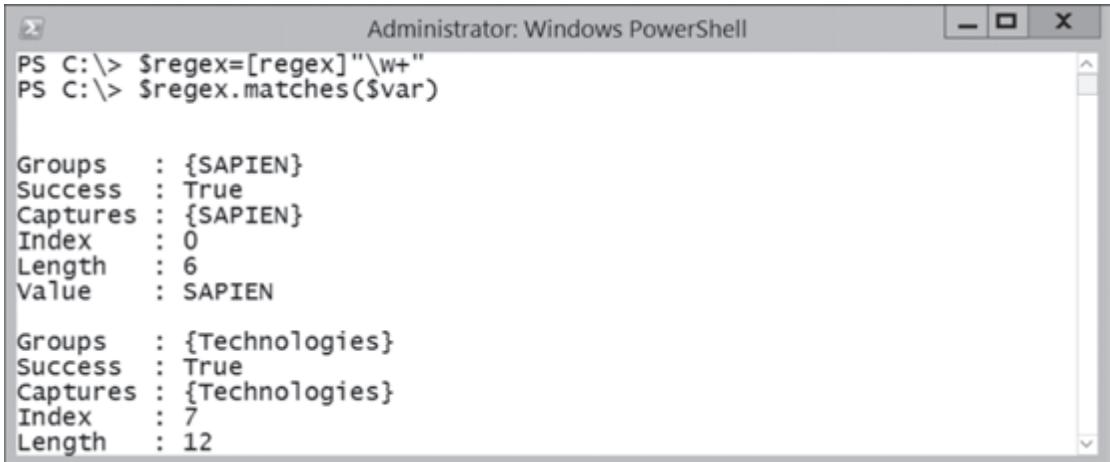
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$var="SAPIEN Technologies PowerShell TFM" is run, followed by PS C:\> \$var -match "\w+". The output shows "True" and then a table titled "\$Matches" with one row. The table has two columns: "Name" and "Value". The "Name" column has a single entry "0" and the "Value" column has a single entry "SAPIEN". The command PS C:\> is shown again at the bottom.

Name	Value
0	SAPIEN

Figure 7-16

To match all instances of the pattern, you need to use the **-Regex** object. In Figure 7-17, notice that the **Matches()** method returns all matches in **\$var**.

```
$regex = [regex]"\w+"
$regex.matches($var)
```



```
Administrator: Windows PowerShell
PS C:\> $regex=[regex]"\w+"
PS C:\> $regex.matches($var)

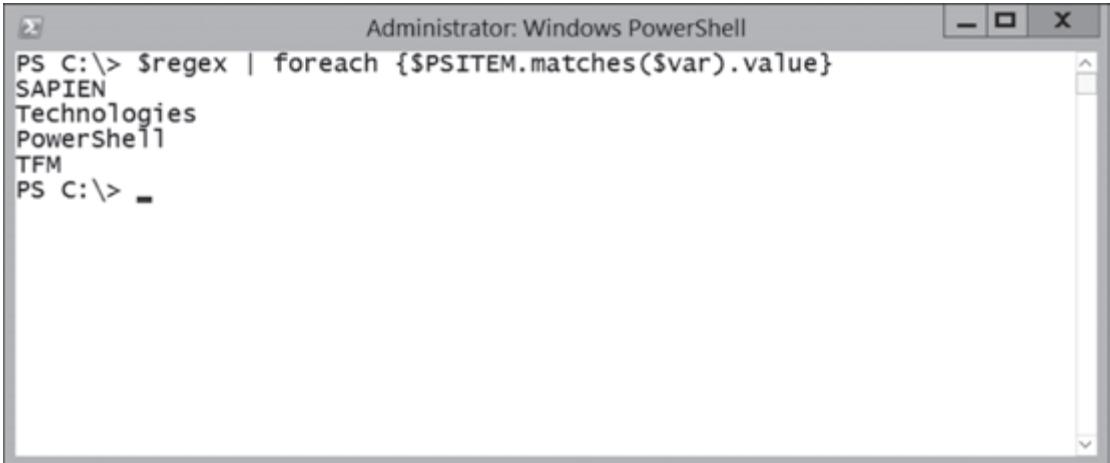
Groups      : {SAPIEN}
Success     : True
Captures   : {SAPIEN}
Index       : 0
Length      : 6
Value       : SAPIEN

Groups      : {Technologies}
Success     : True
Captures   : {Technologies}
Index       : 7
Length      : 12
```

Figure 7-17

We create an object variable called **\$regex** and cast it to a **-Regex** object using **[regex]**, by specifying the regular-expression pattern. We can now call the **Matches()** method of the **-Regex** object, using **\$var** as a parameter. The method returns all instances where the pattern matches, as well as where they were found in **\$var**. A more direct way to see all the matches is to use the **Value** property.

```
$regex | foreach { $PSITEM.matches($var).value }
```

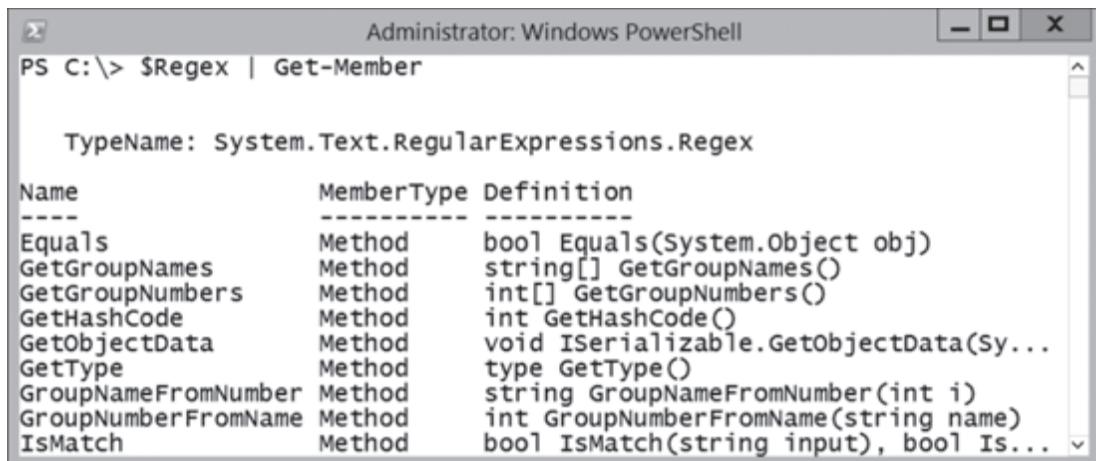


```
Administrator: Windows PowerShell
PS C:\> $regex | foreach {$PSITEM.matches($var).value}
SAPIEN
Technologies
PowerShell
TFM
PS C:\> _
```

Figure 7-18

Using a **Foreach** loop, we can enumerate the collection and display the **Value** property for each item in the collection. The **-Regex** object has several methods and properties that you will want to become familiar with.

```
$Regex|Get-Member
```



```
Administrator: Windows PowerShell
PS C:\> $Regex | Get-Member

TypeName: System.Text.RegularExpressions.Regex

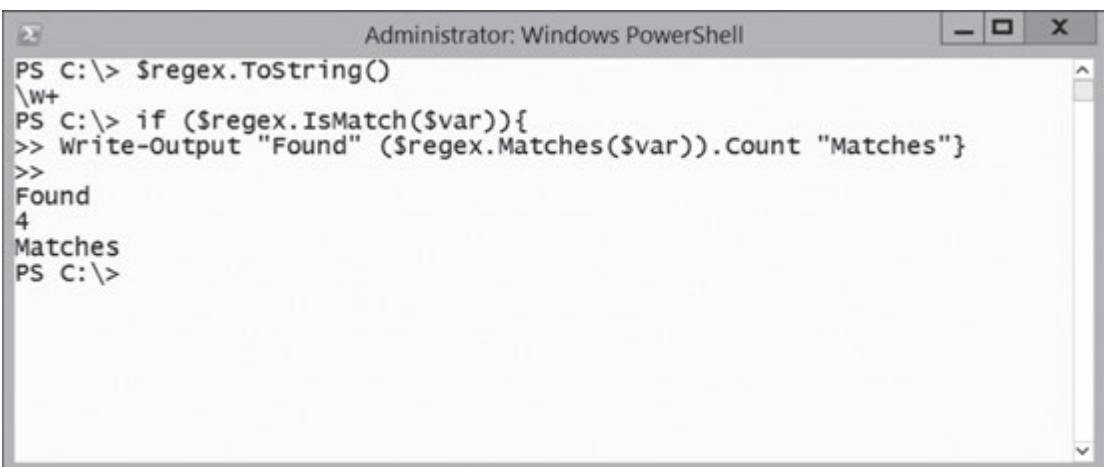
Name           MemberType  Definition
----           --          --
Equals         Method     bool Equals(System.Object obj)
GetGroupNames Method     string[] GetGroupNames()
GetGroupNumbers Method    int[] GetGroupNumbers()
GetHashCode    Method     int GetHashCode()
GetObjectData Method    void ISerializable.GetObjectData(Sy...
GetType        Method     type GetType()
GroupNameFromNumber Method   string GroupNameFromNumber(int i)
GroupNumberFromName Method  int GroupNumberFromName(string name)
IsMatch        Method     bool IsMatch(string input), bool Is...
```

Figure 7-19

In order to see the current value of `$regex`, we need to use the `Tostring()` method.

```
$regex.ToString()

if ($regex.IsMatch($var)) {
Write-Output "Found" ($regex.Matches($var)).Count "Matches" }
```



```
Administrator: Windows PowerShell
PS C:\> $regex.ToString()
\w+
PS C:\> if ($regex.IsMatch($var)){
>> Write-Output "Found" ($regex.Matches($var)).Count "Matches"
>>
Found
4
Matches
PS C:\>
```

Figure 7-20

In Figure 7-20, we check to see if `IsMatch()` is **True**. If it is **True**, PowerShell will display the number of matches found in the string. By the way, the `Count()` method is not a property of the `-Regex` object, but the result of evaluating `$regex.Matches($var)`, which returns a collection object.

```
($regex.Matches($Var)).GetType() | ft -AutoSize
```

IsPublic	IsSerial	Name	BaseType
True	True	MatchCollection	System.Object

Figure 7-21

## Replace operator

You can also use regular expressions to perform a find and replace operation. Simple operations can be done with the **Replace** operator.

```
$text = "The quick brown fox jumped over the lazy cat"
$text = $text -replace "cat", "dog"
$text
```

```
PS C:\> $text="The quick brown fox jumped over the lazy cat"
PS C:\> $text=$text -replace "cat", "dog"
PS C:\> $text
The quick brown fox jumped over the lazy dog
PS C:\> _
```

Figure 7-22

In this example, we've replaced all patterns of "cat" with "dog".

But this example doesn't really take advantage of the **-Regex** object. Consider the example in Figure 7-23.

```
[regex]$regex = "[\s:]"
$c = (get-date).ToString("yyyy-MM-dd_HH-mm-ss")
$c

$d = $regex.replace($c, "_")
$d
```

```
Administrator: Windows PowerShell
PS C:\> [regex]$regex = "[\s:]"
PS C:\> $c = (get-date).ToString("yyyy-MM-dd_HH-mm-ss")
PS C:\> $c
1_11_16_PM
PS C:\> $d = $regex.replace($c, "_")
PS C:\> $d
1_11_16_PM
PS C:\>
```

Figure 7-23

The regular-expression pattern is searching for any space character or colon. We're going to use it against a variable that holds the result of **Get-Date**. The idea is that we want to use the time stamp as a filename, but this means we need to replace the colon character with a legal filename character. For the sake of consistency, we'll replace all instances with the underscore character.

```
$d = $regex.replace($c, "_")
```

You can now use the value of **\$d** as part of a filename.

With regular expressions, it is critical that you are comparing apples to apples. In order for a regular-expression pattern to match, it must match the pattern, but also not match something else. For example, consider the following variable:

```
$a = "Windows 2012 R2 PowerShell v4"
```

Suppose we want to match a number.

```
[regex]$regex = "\d+"
```

The **-Regex** object will match all numbers in \$a.

```
$regex.Matches($a)
```

```
Administrator: Windows PowerShell
PS C:\> $a="Windows 2012 R2 PowerShell v4"
PS C:\> [Regex]$regex="\d+"
PS C:\> $regex.Matches($a)

Groups      : {2012}
Success    : True
Captures   : {2012}
Index      : 8
Length     : 4
Value      : 2012

Groups      : {2}
Success    : True
Captures   : {2}
Index      : 14
```

Figure 7-24

But if the only number we want to match is at the end, then we need a more specific, regular-expression pattern.

```
[regex]$regex = "\d+\$"
$regex.Matches($a)
```

```
Administrator: Windows PowerShell
PS C:\> [regex]$regex="\d+\$"
PS C:\> $regex.Matches($a)

Groups      : {4}
Success    : True
Captures   : {4}
Index      : 28
Length     : 1
Value      : 4

PS C:\> -
```

Figure 7-25

Now, we are only obtaining a match at the end of the string. Let's go through one more example, to drive this point home. Here's a regular-expression pattern that matches a domain credential.

```
[regex]$regex = "\w+\\w+"
```

This will return **True** for expressions like the following example.

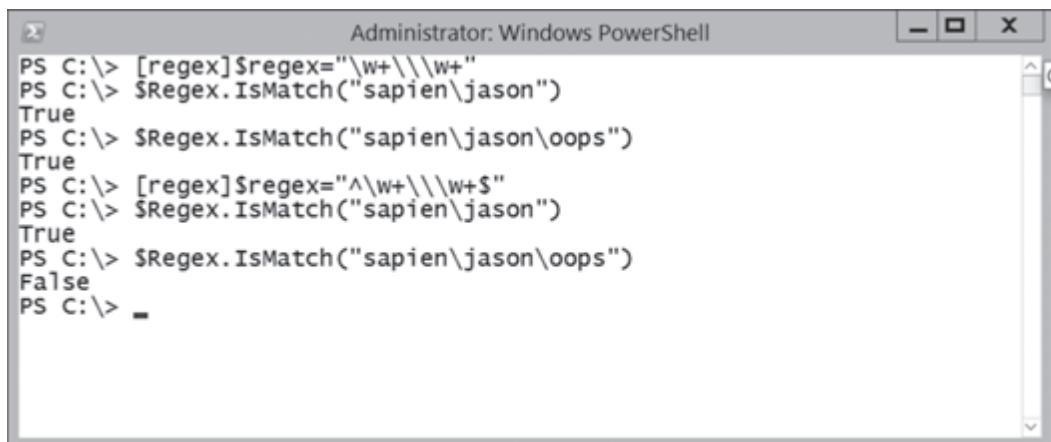
```
$regex.IsMatch("sapien\jason")
```

But it also returns **True** for this:

```
$regex.IsMatch("sapien\jason\oops")
```

Clearly the second string is not a valid credential. To obtain a proper match, we need a regular expression.

```
[regex]$regex = "^\\w+\\\\w+$"
$regex.IsMatch("sapien\jason")
$regex.IsMatch("sapien\jason\oops")
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> [regex]\$regex="\w+\\w+" is run, followed by \$Regex.IsMatch("sapien\jason"), which returns True. Then, \$Regex.IsMatch("sapien\jason\oops") is run, which also returns True. Finally, the command PS C:\> [regex]\$regex="^\\w+\\\\w+\$" is run, followed by \$Regex.IsMatch("sapien\jason"), which returns True. The last command run is PS C:\> \$\_.

Figure 7-26

Now the match is more accurate because the pattern uses `^` to match at the beginning of the string and `$` to match at the end.

## Regular-expression examples

Before we wrap up this quick introduction to regular expressions, let's review the regular expressions that you're most likely to need and use.

### Email address

It's not unreasonable that you might want to search for a string of text that matches an email address pattern. The following is one such regular expression.

```
^([\w-]+)(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$
```

The selection is a sequence consisting of:

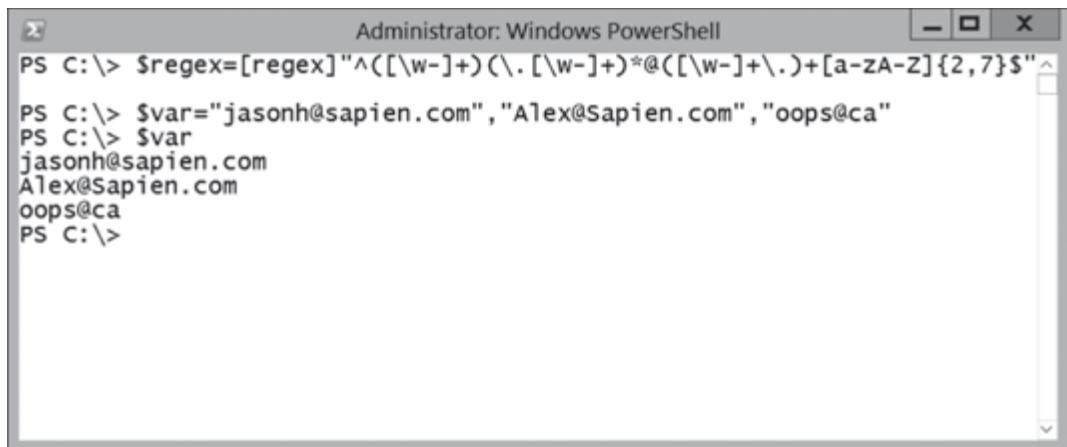
1. A start anchor (^).
2. The expression `([\w-]+)` that matches any word string and the dash character.
3. The expression `(\.[\w-]+)*` that matches a period, and then any word string and the dash.
4. The @ character.
5. The expression `([\w-]+\.)+` that matches any word string that ends in a period.
6. The expression `[a-zA-Z]{2,7}` that matches any string of letters and numbers at least two characters long and no more than seven. This should match domain names like .ca and .museum.
7. An end anchor (\$).

### There's more than one way

There are many, different, regular-expression patterns for an email address. Even though this particular pattern should work for just about any address, it is not 100 percent guaranteed. We used this pattern because it is relatively simple to follow.

The following is how we might use this regular expression.

```
$regex = [regex]"\^([\w-]+)(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}\$"
$var = "jasonh@sapien.com", "Alex@Sapien.com", "oops@ca"
$var
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> $regex=[regex] "^(\\w-)+(.\\w-+)*@([\\w-]+\\.)+[a-zA-Z]{2,7}$"
PS C:\> $var="jasonh@sapien.com", "Alex@Sapien.com", "oops@ca"
PS C:\> $var
jasonh@sapien.com
Alex@Sapien.com
oops@ca
PS C:\>
```

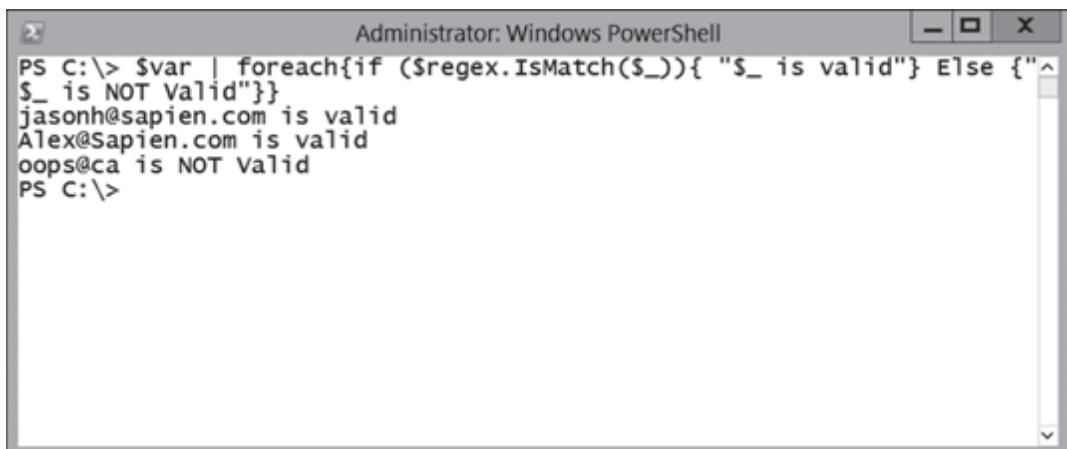
Figure 7-27

We start by creating a **-Regex** object with our email pattern and define an object variable with some email names to check. We've introduced one entry that we know will fail to match. The easiest way to list the matches is to use the **-Match** operator that returns all the valid email addresses.

```
$regex.Matches($var)
$regex.IsMatch($var)
```

If you try expressions like the ones in the previous example, you will see that nothing or **False** is returned. This occurs because **\$var** is an array. We need to enumerate the array and evaluate each element against the regular-expression pattern.

```
$var | foreach{if ($regex.IsMatch($_)){ "$_ is valid"} Else {"$_ is NOT Valid"}}
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> $var | foreach{if ($regex.IsMatch($_)){ "$_ is valid"} Else {"$_ is NOT Valid"}}
PS C:\> $var
jasonh@sapien.com is valid
Alex@Sapien.com is valid
oops@ca is NOT Valid
PS C:\>
```

Figure 7-28

In Figure 7-28, we're enumerating each item in `$var`. If the current variable `$_` matches the regular-expression pattern, then we display a message confirming the match. Otherwise, we display a non-matching message.

## String with no spaces

Up to now, we've been using regular expressions to match alphanumeric characters. However, we can also match whitespaces such as a space, tab, new line, or the lack of whitespace. The following is a `-Regex` object that uses `\S` that is looking to match non-whitespace characters.

```
$regex = [regex]"\S"
$var = "The-quick-brown-fox-jumped-over-the-lazy-dog."
$var2 = "The quick brown fox jumped over the lazy dog."
```

In this example, we have two variables—one with whitespaces and the other without. Which one will return **True** when evaluated with the `IsMatch()` method?

```
$regex.IsMatch($var)
$regex.IsMatch($var2)
```

Actually, this is a trick question, because both return **True**. This happens because `\S` is looking for any non-whitespace character. Since each letter or the dash is a non-whitespace character, the pattern matches. If our aim is to check a string to find out if it contains any spaces, then we really need to use a different regular expression and understand that a finding of **False** is what we're seeking.

```
$regex = [regex]"\s{1}"
$regex.IsMatch($var)
$regex.IsMatch($var2)
```

```
Administrator: Windows PowerShell
PS C:\> $regex=[regex]"\S"
PS C:\> $var="The-quick-brown-fox-jumped-over-the-lazy-dog."
PS C:\> $var2="The quick brown fox jumped over the lazy dog."
PS C:\> $regex.IsMatch($var)
True
PS C:\> $regex=[regex]"\s{1}"
PS C:\> $regex.IsMatch($var)
False
PS C:\> $regex.IsMatch($var2)
True
PS C:\> -
```

Figure 7-29

The regular expression `\s{1}` is looking for a whitespace character that occurs only once. Evaluating `$var` with the `IsMatch()` method returns **False**, because there are no spaces in the string. The same execution with `$var2` returns **True**, because there are spaces in the string. So, if we wanted to take some action based on this type of negative matching, we might use a script like the `NegativeMatchingTest.ps1`.

### **NegativeMatchingTest.ps1**

```
$var = "The-quick-brown-fox-jumped-over-the-lazy-dog."
$var2 = "The quick brown fox jumped over the lazy dog."
$regex = [regex]"\s{1}"
$var
if (($regex.IsMatch($var)) -eq "False")
{
    Write-Output "Expression has spaces"
}
else
{
    Write-Output "Expression has no spaces" }

$var2
if (($regex.IsMatch($var2)) -eq "False")
{
    Write-Output "Expression has spaces"
}
else
{
    Write-Output "Expression has no spaces"

}
```

The purpose of this example is to illustrate that there may be times when you want to match on something that is missing or a negative pattern.

### **Telephone number**

Matching a phone number is pretty straightforward. We can use the pattern `\d{3}-\d{4}` to match any basic phone number, without the area code.

```
$regex = [regex]"\d{3}-\d{4}"
"555-1234" -match $regex
```

This will return **True**, unlike the following which will return **False**.

```
"5551-234" -match $regex
$regex.IsMatch("abc-defg")
```

We hope these examples are looking familiar. First, we defined a regular-expression object, and then we tested different strings to see if there was a match. You can see that only three digits (`\d{3}`) plus a dash (-) plus four digits (`\d{4}`) make a match.

We've only scratched the surface on regular expressions. This is a very complex topic that extends well beyond the scope of this book. Even so, you have had a chance to get your feet wet with the power of regular expressions.



## In Depth 08

# Assignment operators

## Assignment operators

PowerShell uses assignment operators to set values to variables. We've been using the equal sign, but there are many other operators as well. The following table lists PowerShell assignment operators. You can always view the Help file.

```
get-help about_Assignment_Operators
```

**Table 8.1 PowerShell Assignment Operators**

<u>Operator</u>	<u>Description</u>
=	Sets the value of a variable to the specified value.
+=	Increases the value of a variable by the specified value or appends to the existing value.
-=	Decreases the value of a variable by the specified value.
*=	Multiplies the value of a variable by the specified value or appends to the existing value.
/=	Divides the value of a variable by the specified value.
%=	Divides the value of a variable by the specified value and assigns the remainder (modulus) to the variable.

### Windows PowerShell: TFM

In addition to the traditional uses of `=`, in PowerShell this operator has a few extra bells and whistles that might be of interest to you. First, when you assign a hexadecimal value to a variable, it is stored as its decimal equivalent.

```
$var = 0x10  
$var
```

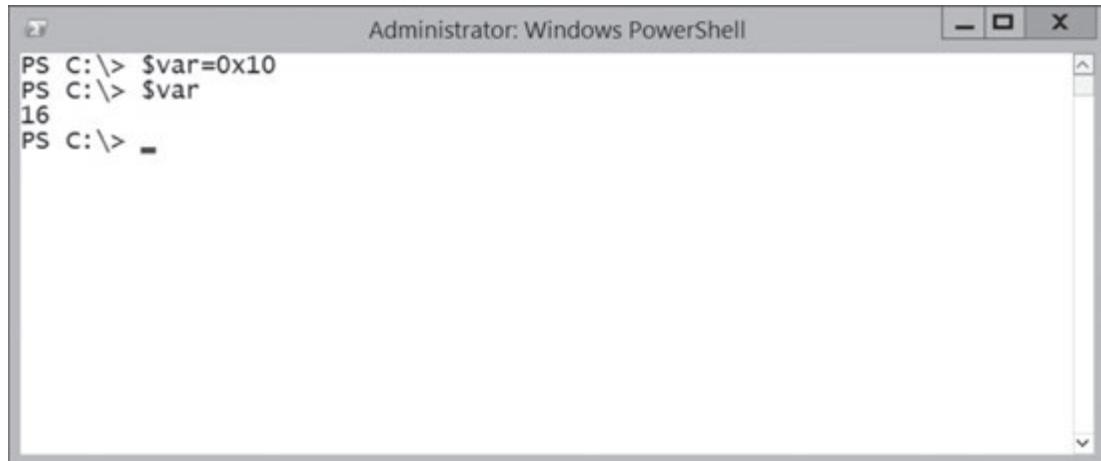
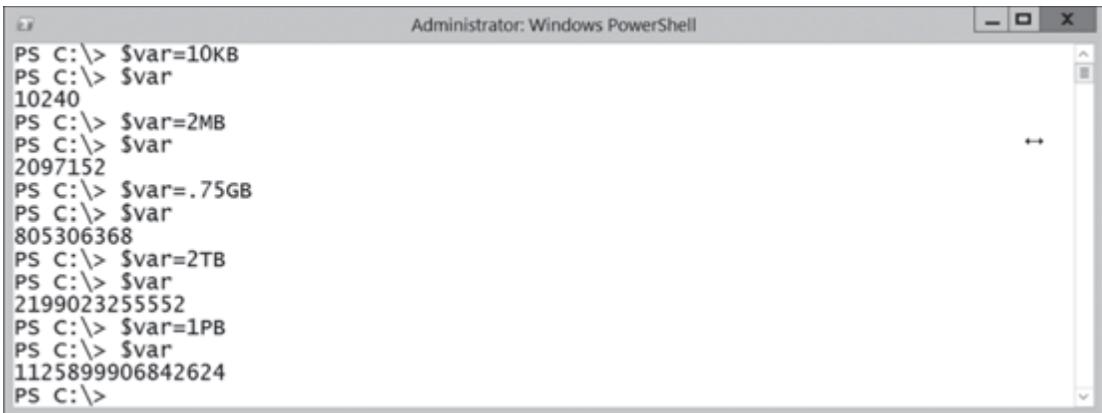


Figure 8-1

You can also use a type of shorthand to assign a variable a multiple byte value. By using KB, MB, GB, TB, and PB, which are known as numeric constants, you store actual kilobyte, megabyte, gigabyte, terabyte, and petabyte values.

```
$var = 10KB  
$var  
  
$var = 2MB  
$var  
  
$var = .75GB  
$var  
  
$var = 2TB  
$var  
  
$var = 1PB  
$var
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$var=10KB is run, followed by PS C:\> \$var, which outputs the value 10240. This pattern repeats for 2MB (2097152), .75GB (805306368), 2TB (2199023255552), and 1PB (1125899906842624).

```

PS C:\> $var=10KB
PS C:\> $var
10240
PS C:\> $var=2MB
PS C:\> $var
2097152
PS C:\> $var=.75GB
PS C:\> $var
805306368
PS C:\> $var=2TB
PS C:\> $var
2199023255552
PS C:\> $var=1PB
PS C:\> $var
1125899906842624
PS C:\>

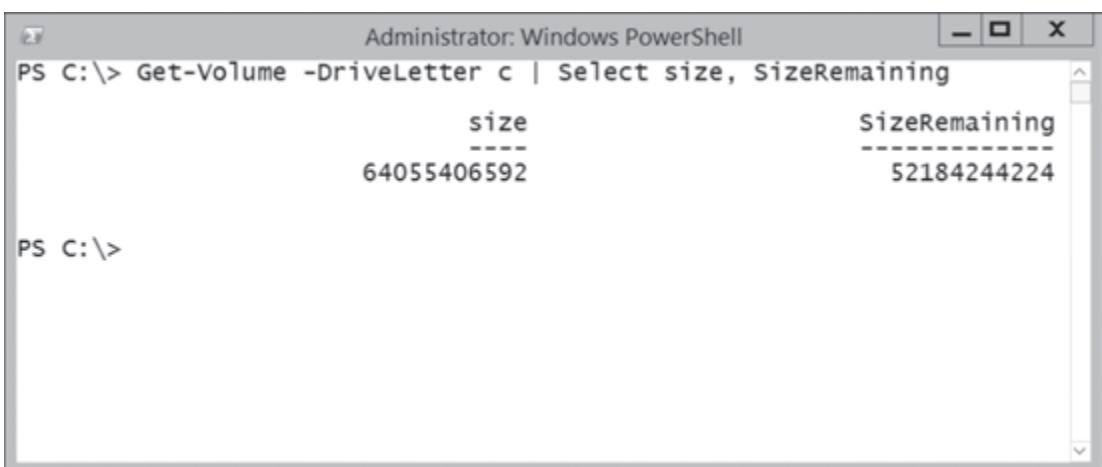
```

Figure 8-2

In the first example, we set **\$var** to 10 KB or 10 kilobytes. Displaying the contents of **\$var** shows the actual byte value of 10 kilobytes. We repeat the process by setting **\$var** to 2 megabytes, .75 gigabytes, 2 terabytes, and 1 petabyte. In each example, we display the value of **\$var**.

Another way to think about using this shorthand is when you receive numeric data that you want to see in MB, GB, TB, etc.

```
Get-Volume -DriveLetter c | Select size, SizeRemaining
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Get-Volume -DriveLetter c | Select size, SizeRemaining is run. The output shows two columns: "size" (64055406592) and "SizeRemaining" (52184244224).

size	SizeRemaining
64055406592	52184244224

```

PS C:\> Get-Volume -DriveLetter c | Select size, SizeRemaining
      size
      ----
      64055406592
      SizeRemaining
      -----
      52184244224
PS C:\>

```

Figure 8-3

As you can tell, the numbers are not very friendly to read. Instead, use a calculated property and change them to something more appealing. How about we display the numbers in gigabytes?

Windows PowerShell: TFM

```
Get-Volume -DriveLetter c | Select DriveLetter,  
@{n = 'Size(GB)'; e = { $PSItem.size / 1gb } },  
@{n = 'Free(GB)'; e = { $PSItem.SizeRemaining / 1gb } }
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Get-Volume -DriveLetter c | Select DriveLetter,  
>> @{n='Size(GB)';e={$PSItem.size / 1gb}},  
>> @{n='Free(GB)';e={$PSItem.SizeRemaining /1gb}}  
>>
```

The resulting table output is:

DriveLetter	Size(GB)	Free(GB)
C	59.6562461853027	48.6003646850586

PS C:\>

Figure 8-4

That still isn't particularly appealing, really. So how about we also round the numbers and eliminate the long decimal points?

```
Get-Volume -DriveLetter c | Select DriveLetter,  
@{n = 'Size(GB)'; e = { $PSItem.size / 1gb -as [Int] } },  
@{n = 'Free(GB)'; e = { $PSItem.SizeRemaining / 1gb -as [Int] } }
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Get-Volume -DriveLetter c | Select DriveLetter,  
>> @{n='Size(GB)';e={$PSItem.size / 1gb -as [Int]}},  
>> @{n='Free(GB)';e={$PSItem.SizeRemaining /1gb -as [Int]}}  
>>
```

The resulting table output is:

DriveLetter	Size(GB)	Free(GB)
C	60	49

PS C:\>

Figure 8-5

That's starting to look pretty good!

The `+=` operator increases the value of a given variable by a specified amount.

```
$var = 7
$var
7
```

```
$var += 3
$var
10
```

The variable `$var` begins with a value of 7. We then used `+=` to increment it by 3, which changed the value of `$var` to 10.

The `-=` operator decreases the value of a given variable by a specified amount. Let's build on the previous example.

```
$var -= 3
$var
7
```

`$var` starts out with a value of 10. Using the `-=` operator, we decreased its value by 3, which returned us to 7.

What if we want to multiply a variable value by a specific number? This calls for the `*=` operator. Let's continue with the same `$var` that currently has a value of 7.

```
$var *= 3
$var
21
```

We can also divide by using the `/=` operator.

```
$var /= 7
$var
3
```

Finally, we can use `%=` to divide the variable value by the assigned value and return the modulus or remainder.

```
$var = 9
$var %= 4
$var
1
```

In this example, we start with a `$var` value of 9. Using the modulus assignment operator with a value of 4 means we're dividing 9 by 4. The remainder value is then assigned to `$var`, which in this example is 1.

You need to be careful with assignment operators and variable values. Remember, PowerShell does a pretty good job at deciding if what you typed is a number or a string. If you put something in quotes, PowerShell treats it as a string. If you're not careful, you can receive some odd results.

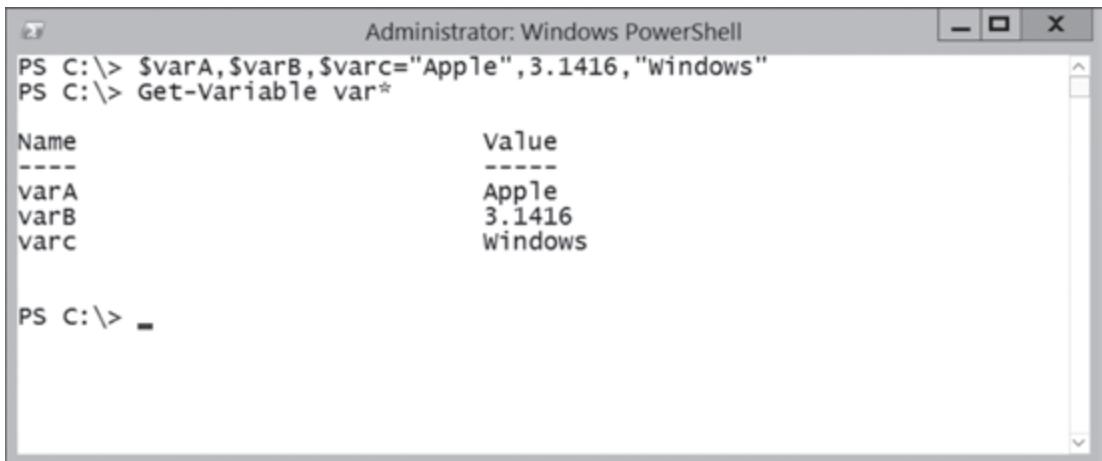
```
$var = "3"
$var += 7
$var
37
```

In this example, we think we set `$var` to 3 and increased it by 7 using the `+=` operator. However, "3" is a string, so the `+=` operator simply concatenates, instead of adds, which is why we end up with a `$var` value of 37. If you are ever unsure about what type of object you're working with, you can use the `GetType()` method.

```
$var.GetType()
```

It is possible to assign values to multiple variables with a single statement.

```
$varA, $varB, $varC = "Apple", 3.1416, "Windows"
Get-Variable var*
```



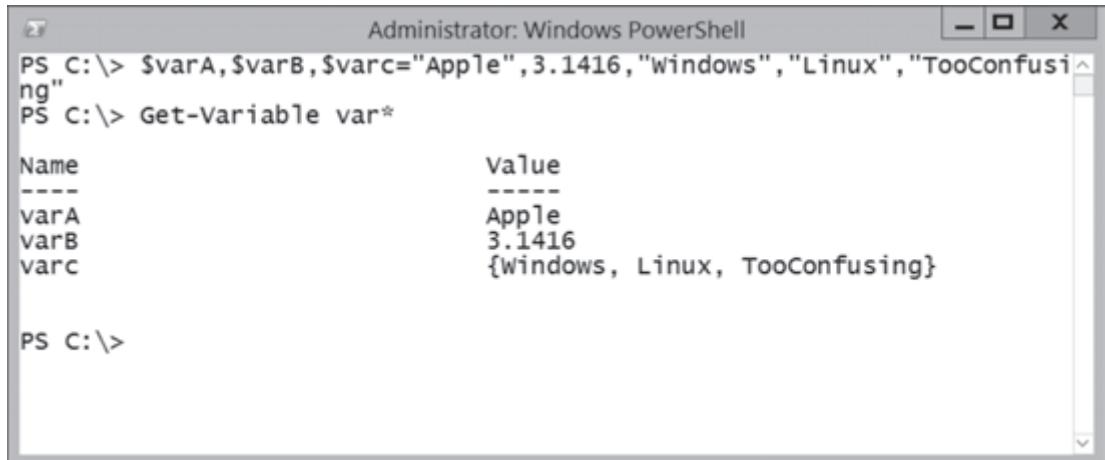
The screenshot shows an Administrator Windows PowerShell window. The command `$varA, $varB, $varC = "Apple", 3.1416, "Windows"` is entered, followed by `Get-Variable var*`. The output displays three variables:

Name	Value
varA	Apple
varB	3.1416
varC	Windows

Figure 8-6

The assigned values are set to their respective variables. If you have more values than variables, then the extra values are assigned to the last variable.

```
$varA, $varB, $varc = "Apple", 3.1416, "Windows", "Linux", "TooConfusing"  
Get-Variable var?
```



The screenshot shows an Administrator Windows PowerShell window. The command `$varA, $varB, $varc = "Apple", 3.1416, "Windows", "Linux", "TooConfusing"` is entered, followed by `Get-Variable var*`. The output displays a table:

Name	Value
varA	Apple
varB	3.1416
varc	{Windows, Linux, TooConfusing}

PS C:\>

Figure 8-7

Our recommendation is to be careful with this type of statement, because you can end up with unintentional variable values. PowerShell will wait—so set and modify variables.



## In Depth 09

# Managing Active Directory with ADSI, Part 1

## Using Active Directory with ADSI

Before we get started, we need to make sure you have some realistic expectations about Active Directory Service Interfaces (ADSI) in Windows PowerShell. First, understand that ADSI support was always planned to be a part of PowerShell. Since PowerShell is built on the .NET Framework, it can access the underlying directory service classes. The PowerShell team has provided some type adapters to make working with these classes much easier.

The Active Directory (AD) team has also released a PowerShell solution for managing AD through the Active Directory Module. These cmdlets work with AD from the current server release back to Windows 2003 domain controllers. While we generally prefer to use these cmdlets for most of our AD needs, sometimes you just need to go a little deeper. Also, you may already have several scripts using ADSI and may need to maintain them. While this chapter focuses on ADSI, if you want to dig into the cmdlets for AD, see SAPIEN's TFM "Managing Active Directory with Windows PowerShell" by Jeffery Hicks.

## ADSI fundamentals

Active Directory Service Interfaces is an extremely misleading name. Rather than reading it as "**Active Directory** Service Interfaces," which is what most admins think it is; you should think of it as "Directory Service Interfaces." ADSI was named at a time when Microsoft slapped the word "Active" on everything that wasn't bolted down: ActiveX Data Objects, Active Directory, Active Documents,

and more. The thing to remember, though, is that ADSI isn't just for Active Directory. It works great with old Windows NT 4.0 domains and even works with the local security accounts on standalone and member computers running modern versions of Windows.

ADSI is built around a system of providers. Each provider is capable of connecting to a particular type of directory: Windows NT (which includes local security accounts on standalone and member computers), Lightweight Directory Access Protocol (LDAP—this is what Active Directory uses), and even Novell Directory Services, if you still have that in your environment somewhere.

## ADSI queries

The primary way to access directory objects—that is, users, groups, and so forth—is by issuing an ADSI query. A query starts with an ADSI provider, so that ADSI knows which type of directory you're trying to talk to. The two providers you'll use most are WinNT:// and LDAP://—and note that, unlike most things in PowerShell, these provider names are case sensitive, and that they use forward slashes, not backslashes. Those two caveats mess us up every time!

The format of an ADSI query depends on the provider. For the WinNT:// provider, here is what a query looks like.

```
WinNT://NAMESPACE/OBJECT,CLASS
```

The **NAMESPACE** portion of the query can either be a computer name or a NetBIOS domain name—including AD domains! Remember that AD is backward-compatible with Windows NT, and by using the WinNT:// provider to access AD, you'll be able to refer to directory objects—users and groups—without needing to know what organizational unit (OU) they're in, because Windows NT didn't have OUs. The **OBJECT** portion of the query is the object name—that is, the user name, group name, or whatever—that you're after. The class part of the query is technically optional, but we recommend always including it. It should be “user” if you're querying a user object, “group” for a group object, and so forth. So, here's how a complete query for our test machine's local administrator account would look like.

```
WinNT://Server/Administrator,user
```

An LDAP query is much different. These queries require a distinguished name including the ADSI provider. For example, if you need to get the SalesUsers group, which is in the East OU, which is in the Sales OU of the MyDomain.com domain, here is what your query would look like.

```
LDAP://cn=SalesUsers,ou=East,ou=Sales,dc=MyDomain,dc=com
```

Definitely a bit more complicated, but this syntax should be almost second nature to an AD manager. LDAP queries don't directly support wildcards either—you need to know exactly which object you're after. PowerShell does provide a somewhat cumbersome .NET Framework-based means of searching for directory objects, which we'll look at.

## Using ADSI objects

Once you've queried the correct object, you can work with its properties and methods. Objects queried through the WinNT:// provider generally have several useful properties. Although, be aware that if you're accessing an AD object through the WinNT:// provider, you won't have access to all of the object's properties. You'll only see the ones that the older WinNT:// provider understands. A few methods are available, too, such as **SetPassword()** and **SetInfo()**. The **SetInfo()** method is especially important—you must execute it after you change any object properties, so that the changes you made will be saved back to the directory correctly.

Objects retrieved through the LDAP:// provider don't directly support many properties. Instead, you execute the **Get()** and **GetEx()** methods, passing the property name you want, to retrieve properties. For example, assuming the variable **\$user** represented a user object, here is how you would retrieve the **Description** property.

```
$user.Get("Description")
```

**Get()** is used to retrieve properties that have only a single value, such as **Description**. **GetEx()** is used for properties that can contain multiple values, such as AD's **otherHomePhone**. The opposites of these two methods are **Put()** and **PutEx()**.

```
$user.Put("Description", "New Value")
```

After you finish all the **Put()** and **PutEx()** calls you want, you must execute the **SetInfo()** method to save the changes back to the directory. As with the WinNT:// provider, security principals retrieved through the LDAP:// provider also have a **SetPassword()** method that you can use.

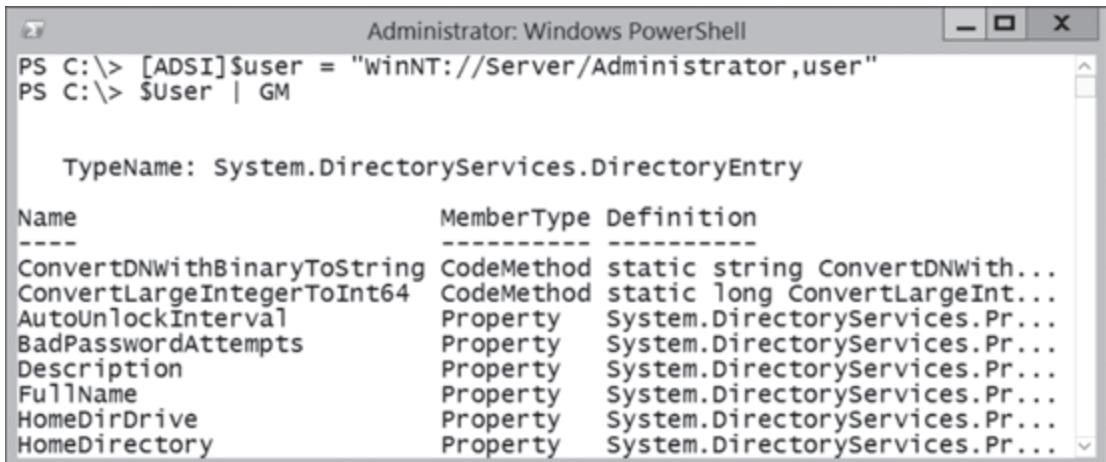
## Retrieving ADSI objects

If you're not using the cmdlets from the Active Directory (AD) module, then you will use the **[ADSI]** type accelerator to retrieve objects. You'll need to start with an ADSI query string, which we showed you how to build in the “ADSI Queries” section. Then, just feed that to the type accelerator.

```
[ADSI]$user = "WinNT://Server/Administrator,user"
```

This will retrieve the local Administrator user account from the computer named Server, by using the WinNT:// provider. You can then pipe the resulting object—which we've stored in the **\$user** variable—to **Get-Member** (or its alias, **GM**) to see what properties and methods the object contains.

```
$user | GM
```



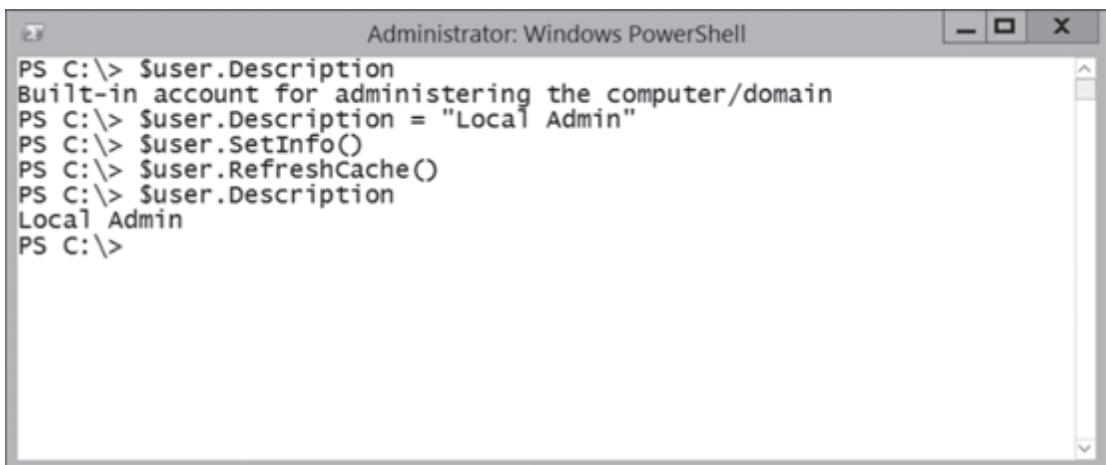
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> [ADSI]\$user = "WinNT://Server/Administrator,user" is run, followed by PS C:\> \$User | GM. The output displays the properties of the user object:

Name	MemberType	Definition
ConvertDNWithBinaryToString	CodeMethod	static string ConvertDNwith...
ConvertLargeIntegerToInt64	CodeMethod	static long ConvertLargeInt...
AutoUnlockInterval	Property	System.DirectoryServices.Pr...
BadPasswordAttempts	Property	System.DirectoryServices.Pr...
Description	Property	System.DirectoryServices.Pr...
FullName	Property	System.DirectoryServices.Pr...
HomeDirDrive	Property	System.DirectoryServices.Pr...
HomeDirectory	Property	System.DirectoryServices.Pr...

Figure 9-1

We started with a WinNT:// provider example because these are perhaps the easiest objects to work with, and you will receive nice, clearly defined properties. However, remember in the “Using ADSI Objects” section, we said you have to execute the object’s **SetInfo()** method whenever you change any properties? Do you see the **SetInfo()** method listed above? Nope. And that’s because a major problem with the **[ADSI]** type accelerator is that it doesn’t pass in the object’s methods—only its properties. You can still use the **SetInfo()** method, though.

```
$user.Description = "Local Admin"
$user.SetInfo()
$user.RefreshCache()
$user.Description
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The commands demonstrate changing the user's description:

```
PS C:\> $user.Description
Built-in account for administering the computer/domain
PS C:\> $user.Description = "Local Admin"
PS C:\> $user.SetInfo()
PS C:\> $user.RefreshCache()
PS C:\> $user.Description
Local Admin
PS C:\>
```

Figure 9-2

It's just that the method doesn't show up in **Get-Member**, so you'll have to remember the method on your own. Basically, though, that's how you work with objects from the WinNT:// provider: Query the object, view or modify properties, and call **SetInfo()** if you've changed any properties. Use **SetPassword()** to change the password of a user object.

Although it isn't shown in the output of **Get-Member**, you can also use the **Get()**, **Put()**, **GetEx()**, and **PutEx()** methods we discussed in the "Using ADSI Objects" section.

```
$user.get("Description")
```

This isn't really useful with local computer accounts, since the object has direct properties you can access.

Here, you can see the WinNT:// provider being used to access an AD user named JasonH from the COMPANY domain (note that you have to use the domain's "short," or NetBIOS name, not its full DNS name).

```
[ADSI]$user = "WinNT://COMPANY/JasonH,user"
$user | gm -MemberType Property
```

Name	MemberType	Definition
AutoUnlockInterval	Property	System.DirectoryServices.Pro...
BadPasswordAttempts	Property	System.DirectoryServices.Pro...
Description	Property	System.DirectoryServices.Pro...
FullName	Property	System.DirectoryServices.Pro...
HomeDirDrive	Property	System.DirectoryServices.Pro...
HomeDirectory	Property	System.DirectoryServices.Pro...
LockoutObservationInterval	Property	System.DirectoryServices.Pro...
LoginHours	Property	System.DirectoryServices.Pro...

Figure 9-3

So, where are all the AD-specific properties, like **otherHomePhone** and **SN**? Well, perhaps we could use the **Get()** method to retrieve one of them.

```
$user.GET("sn")
```

```

Administrator: Windows PowerShell
PS C:\> $user.GET("sn")
Exception calling "GET" with "1" argument(s): "The directory
property cannot be found in the cache."
At line:1 char:1
+ $user.GET("sn")
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodInvocation
Exception
+ FullyQualifiedErrorId : CatchFromBaseAdapterMethodInvokeTI
PS C:\>

```

Figure 9-4

Nope. It turns out that the WinNT:// provider can't see any additional properties from AD; it can only see those properties that are backward-compatible with Windows NT 4.0 domains. So, when you're using the WinNT:// provider to access AD, you're giving up a lot of AD's extended capabilities.

Which brings us to AD's native provider, LDAP://. You'll retrieve objects pretty much the same way as you did for the WinNT:// provider: Use the [ADSI] type accelerator, and provide an LDAP query string.

```
[ADSI]$Domain = "LDAP://dc=company,dc=loc"
$domain | gm -MemberType Property
```

```

Administrator: Windows PowerShell
PS C:\> [ADSI]$Domain="LDAP://dc=company,dc=loc"
PS C:\> $domain | gm -MemberType Property

TypeName: System.DirectoryServices.DirectoryEntry

```

Name	MemberType	Definition
auditingPolicy	Property	System.DirectoryService...
creationTime	Property	System.DirectoryService...
dc	Property	System.DirectoryService...
distinguishedName	Property	System.DirectoryService...
dsASignature	Property	System.DirectoryService...
dsCorePropagationData	Property	System.DirectoryService...
forceLogoff	Property	System.DirectoryService...
fSMORoleOwner	Property	System.DirectoryService...

Figure 9-5

We've truncated the results a bit, but you can see that we've retrieved the domain object and displayed its properties—but not methods, because those won't be shown—by using the LDAP:// provider and the **Get-Member** cmdlet. We can retrieve the built-in **Users** container in a similar fashion.

```
[ADSI]$Container = "LDAP://cn=users,dc=company,dc=loc"
$container | gm -MemberType Property
```

Name	MemberType	Definition
cn	Property	System.DirectoryServices.Property
description	Property	System.DirectoryServices.Property
distinguishedName	Property	System.DirectoryServices.Property
dsCorePropagationData	Property	System.DirectoryServices.Property
instanceType	Property	System.DirectoryServices.Property
isCriticalSystemObject	Property	System.DirectoryServices.Property
name	Property	System.DirectoryServices.Property
nTSecurityDescriptor	Property	System.DirectoryServices.Property

Figure 9-6

Notice anything similar about the container and the domain? They're both **System.DirectoryServices.DirectoryEntry** objects, even though they're very different objects. This is one of the things that makes PowerShell's current ADSI support a bit complicated. PowerShell relies on this underlying .NET Framework class, **DirectoryEntry**, to represent all directory objects. Obviously, different types of objects—containers, users, groups, and so forth—have different properties and capabilities, but this class represents them all generically. PowerShell and the .NET Framework try to represent the object's properties as best they can, but they can't always show you everything that's available. This becomes especially apparent when you view the untruncated output of **Get-Member** for an AD user object.

```
[ADSI]$user = "LDAP://cn=Jason Helmick,ou=IT,dc=company,dc=loc,cn=users"
$user | gm
```

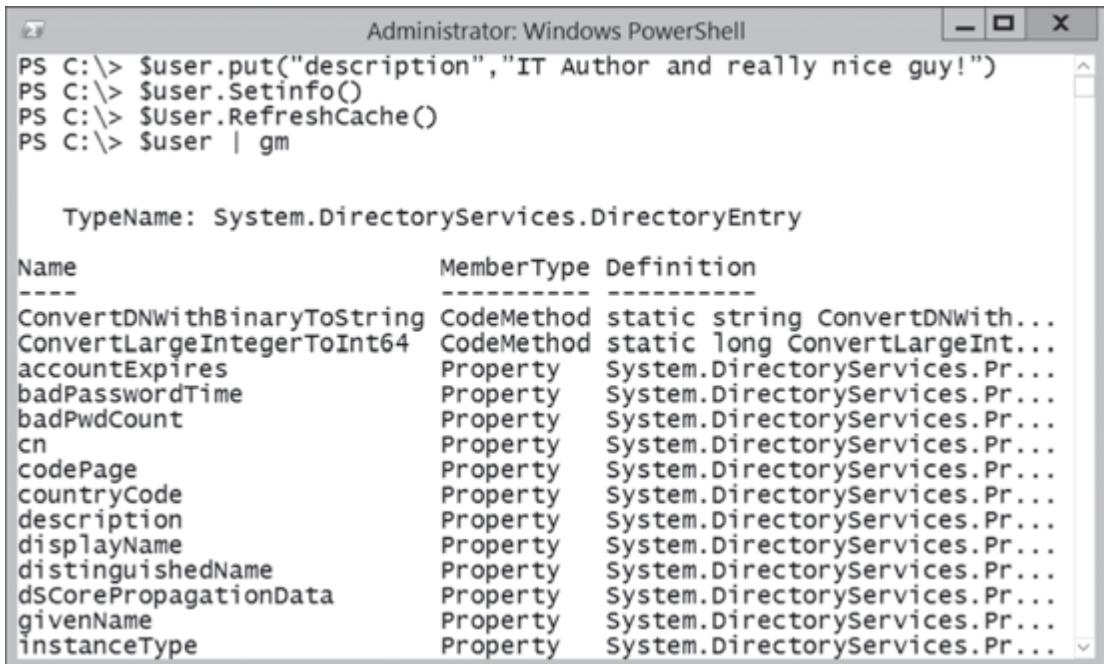
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "[ADSI]\$user = \"LDAP://cn=Jason Helmick,ou=IT,dc=company,dc=local\"; \$user | gm". The output displays the properties and methods of the \$user variable, which is of type System.DirectoryServices.DirectoryEntry. The properties listed include accountExpires, badPasswordTime, badPwdCount, cn, codePage, countryCode, displayName, distinguishedName, dSCorePropagationData, givenName, instanceType, lastLogoff, and lastLogon. Each property is shown with its name, member type (e.g., Property or CodeMethod), and definition.

Name	MemberType	Definition
ConvertDNwithBinaryToString	CodeMethod	static string ConvertDNwith...
ConvertLargeIntegerToInt64	CodeMethod	static long ConvertLargeInt...
accountExpires	Property	System.DirectoryServices.Pr...
badPasswordTime	Property	System.DirectoryServices.Pr...
badPwdCount	Property	System.DirectoryServices.Pr...
cn	Property	System.DirectoryServices.Pr...
codePage	Property	System.DirectoryServices.Pr...
countryCode	Property	System.DirectoryServices.Pr...
displayName	Property	System.DirectoryServices.Pr...
distinguishedName	Property	System.DirectoryServices.Pr...
dSCorePropagationData	Property	System.DirectoryServices.Pr...
givenName	Property	System.DirectoryServices.Pr...
instanceType	Property	System.DirectoryServices.Pr...
lastLogoff	Property	System.DirectoryServices.Pr...
lastLogon	Property	System.DirectoryServices.Pr...

Figure 9-7

Active Directory Users and Computers uses a dialog box to display user properties and it definitely displays more than these! For example, where is the **Description** property? Well, it turns out that the particular user we retrieved doesn't have a **Description** property—that is, it was never filled in when the user was created. So, the property isn't shown. We can set the property—provided we know the property name already, since **Get-Member** won't show it to us. We'll set the property, use **SetInfo()** to save the change, and then re-query the user to see if the property shows up.

```
$user.put("description", "IT Author and really nice guy!")
$user.Setinfo()
$user.RefreshCache()
$user | gm
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$user.put("description","IT Author and really nice guy!") is run, followed by \$user.Setinfo(), \$User.RefreshCache(), and \$user | gm. The output shows the TypeName: System.DirectoryServices.DirectoryEntry and a table of its members:

Name	MemberType	Definition
ConvertDNWithBinaryToString	CodeMethod	static string ConvertDNWith...
ConvertLargeIntegerToInt64	CodeMethod	static long ConvertLargeInt...
accountExpires	Property	System.DirectoryServices.Pr...
badPasswordTime	Property	System.DirectoryServices.Pr...
badPwdCount	Property	System.DirectoryServices.Pr...
cn	Property	System.DirectoryServices.Pr...
codePage	Property	System.DirectoryServices.Pr...
countryCode	Property	System.DirectoryServices.Pr...
description	Property	System.DirectoryServices.Pr...
displayName	Property	System.DirectoryServices.Pr...
distinguishedName	Property	System.DirectoryServices.Pr...
dsCorePropagationData	Property	System.DirectoryServices.Pr...
givenName	Property	System.DirectoryServices.Pr...
instanceType	Property	System.DirectoryServices.Pr...

Figure 9-8

As you can see, the **Description** property now appears, because it has a value. This is an important caveat of working with ADSI in PowerShell: you can't rely on **Get-Member** to discover objects' capabilities. Instead, you'll need an external reference, such as the MSDN documentation.

## Searching for ADSI objects

Sometimes you need to retrieve an object from Active Directory (AD) without knowing exactly where it is or its **distinguishedname**. PowerShell relies on the .NET Framework and the **DirectoryServices.DirectorySearcher** class. This type of object is used to find objects in a directory service such as AD. Here's a sample function that uses the class to find a user object in AD based on the user's SAM account name.

### Find-User Function.ps1

```
Function Find-User {
    Param ($sam = $(throw "you must enter a SAMAccountName"))
    $searcher = New-Object DirectoryServices.DirectorySearcher
    $searcher.Filter = "(&(objectcategory = person)(objectclass = user)(SAMAccountName = $sam))"
    $results = $searcher.FindOne()
    if ($results.path.length -gt 1)
    {
        Write $results
    }
    else
```

```
{
    write "Not Found"
}
}
```

You use the **New-Object** cmdlet to create the **DirectorySearcher** object.

```
$searcher = New-Object DirectoryServices.DirectorySearcher
```

By default, the searcher will search the current domain, although you can specify a location, such as an OU, which we'll show you how to do in a little bit. What you will need to do, however, is specify an LDAP search filter.

```
$searcher.Filter = "(&(objectcategory = person)(objectclass = user)(SAMAccountName = $sam))"
```

The filter instructs the searcher to find user objects where the **SAMAccountName** property matches that passed as a function parameter. The function calls the searcher's **FindOne()** method.

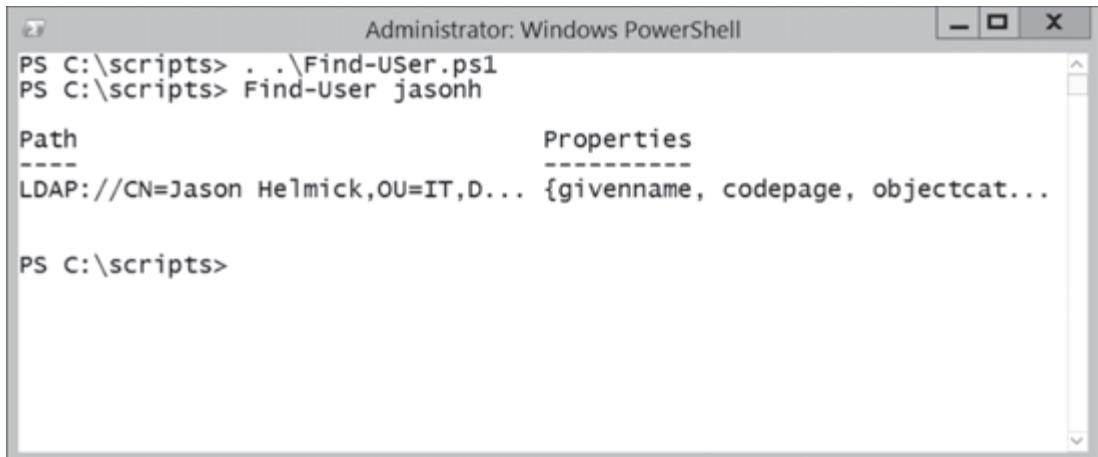
```
$results = $searcher.FindOne()
```

Assuming a user is found, the resulting object will be stored in **\$results**. The script checks the length of the **Path** property of **\$results**. If a user object was found, the **Path** property will be the user's **DistinguishedName** and will have a length greater than 1. Otherwise, the user was not found and the function returns an error message.

```
if ($results.path.length -gt 1)
{
    write $results
}
else
{
    write "Not Found"
}
```

Here's how you can use the function.

```
Find-User jasonh
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `PS C:\scripts> . .\Find-User.ps1` is run, followed by `PS C:\scripts> Find-User jasonh`. The output displays the user's distinguished name (Path) and a properties object (Properties). The Path is shown as "LDAP://CN=Jason Helmick,OU=IT,D...". The Properties object is a collection of properties, with the first few being {givenname, codepage, objectcat...}.

```

Administrator: Windows PowerShell
PS C:\scripts> . .\Find-User.ps1
PS C:\scripts> Find-User jasonh

Path                               Properties
-----
LDAP://CN=Jason Helmick,OU=IT,D... {givenname, codepage, objectcat...}

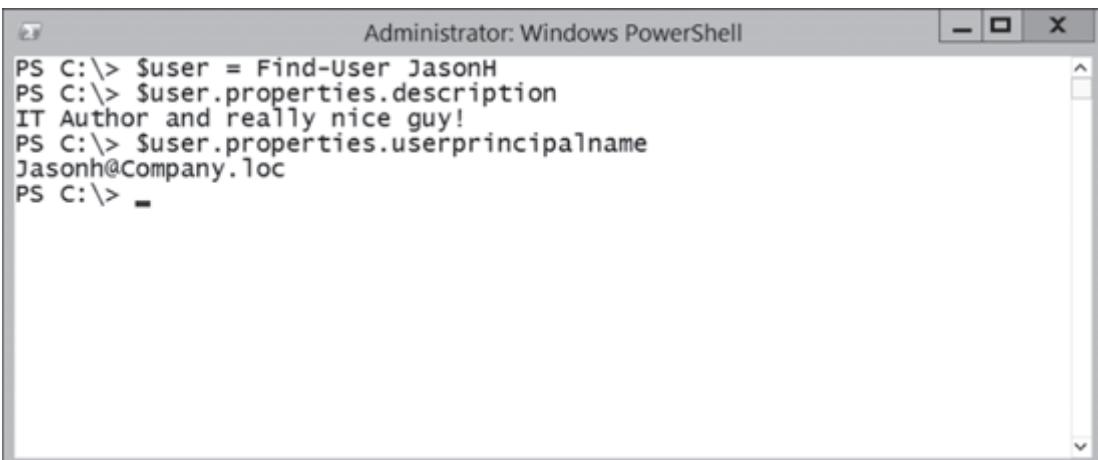
PS C:\scripts>

```

Figure 9-9

The **Path** property shows the user's distinguished name. The **Properties** property is a collection of all the user properties. Here's another way you might use this function.

```
$user = Find-User JasonH
$user.properties.description
$user.properties.userprincipalname
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". A script block is run, starting with `PS C:\> \$user = Find-User JasonH`. Inside the block, `PS C:\> \$user.properties.description` is run, displaying the value "IT Author and really nice guy!". Then, `PS C:\> \$user.properties.userprincipalname` is run, displaying the value "jasonh@Company.loc". Finally, `PS C:\>` is shown.

```

Administrator: Windows PowerShell
PS C:\> $user = Find-User JasonH
PS C:\> $user.properties.description
IT Author and really nice guy!
PS C:\> $user.properties.userprincipalname
jasonh@Company.loc
PS C:\>

```

Figure 9-10

The results of the **Find-User** function are stored in the **\$user** variable. This means that we can access its properties directly, such as **Description** and **UserPrincipalName**. Here's how to see all of user's defined properties.

```
$user.properties
```

You can also use the **Searcher** object to search from a specific container and to find more than one object.

```
$Searcher = New-Object DirectoryServices.DirectorySearcher
$Root = New-Object DirectoryServices.DirectoryEntry ` 
'LDAP://OU=Sales,OU=Employees,DC=mycompany,DC=local'

$Searcher.SearchRoot = $Root
$searcher.Filter = "(&(objectcategory = person)(objectclass = user))"
$searcher.pagesize = 100
$Searcher.FindAll()
```

In the following example, we create a new object type called a **DirectoryEntry**.

```
$Root = New-Object DirectoryServices.DirectoryEntry ` 
'LDAP://OU=Sales,OU=Employees,DC=mycompany;DC=local'
```

You can use this object for the **Root** property of the **Searcher** object.

```
$Searcher.SearchRoot = $Root
```

Again, we're going to search for **User** objects.

```
$searcher.Filter = "(&(objectcategory = person)(objectclass = user))"
```

Only this time, we'll use the **FindAll()** method to return all objects that match the search pattern.

```
$Searcher.FindAll()
```

Now all of that may seem like a lot of work and, frankly, it is. However, we wanted you to know how the **DirectorySearcher** class works so that you can appreciate and understand the **[ADSISeacher]** type adapter.

This type adapter simplifies this process.

```
[ADSISeacher]$searcher = "(&(objectcategory = person)(objectclass = user))"
$searcher | gm
```

```
Administrator: Windows PowerShell
PS C:\> [ADSISearcher]$searcher="(&(objectcategory=person)(objectclass=user))"
PS C:\> $searcher | gm

TypeName: System.DirectoryServices.DirectorySearcher

Name           MemberType  Definition
----           -----      -----
Disposed       Event      System.EventHandler Disposed(...)
CreateObjRef  Method     System.Runtime.Remoting.ObjRe...
Dispose       Method     void Dispose(), void IDisposa...
Equals        Method     bool Equals(System.Object obj)
FindAll        Method     System.DirectoryServices.Sear...
FindOne        Method     System.DirectoryServices.Sear...
GetHashCode   Method     int GetHashCode()
```

Figure 9-11

We generally create the **Searcher** object by defining its filter. The search root defaults to the current domain naming context.

```
$searcher.searchroot
```

```
Administrator: Windows PowerShell
PS C:\> $searcher.SearchRoot

distinguishedName : {DC=Company,DC=loc}
Path              : LDAP://DC=Company,DC=loc

PS C:\> _
```

Figure 9-12

Of course, you could easily change that. The default search scope is **Subtree**, as well.

```
$searcher.SearchScope
```

All that you have to do is call either the **FindOne()** or **FindAll()** method.

```
$searcher.Findall()
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$searcher.SearchScope Subtree was run, followed by PS C:\> \$searcher.Findall(). The output displays a table with two columns: "Path" and "Properties". The "Path" column lists four LDAP paths: LDAP://CN=Administrator,CN=User..., LDAP://CN=Guest,CN=Users,DC=Com..., LDAP://CN=krbtgt,CN=Users,DC=Co..., and LDAP://CN=Jason Helmick,OU=IT,D.... The "Properties" column shows the properties for each object, which are partially cut off at the end. The entire output is as follows:

```
PS C:\> $searcher.SearchScope
Subtree
PS C:\> $searcher.Findall()

Path          Properties
-----
LDAP://CN=Administrator,CN=User... {logoncount, codepage, objectca...
LDAP://CN=Guest,CN=Users,DC=Com... {logoncount, codepage, objectca...
LDAP://CN=krbtgt,CN=Users,DC=Co... {logoncount, codepage, objectca...
LDAP://CN=Jason Helmick,OU=IT,D... {givenname, codepage, objectcat...
```

Figure 9-13

We can now slightly rewrite our original **Find-User** function to take advantage of the type adapter.

### Find-User Revised.ps1

```
Function Find-User {
    Param ($sam = $(throw "you must enter a SAMAccountName"))

    [ADSI searcher] $searcher = "(&(objectcategory = person)(objectclass = user)(SAMAccountName = $sam))"
    $results = $searcher.FindOne()

    if ($results.path.length -gt 1)
    {
        write $results
    }
    else
    {
        write "Not Found"
    }
}
```

## Working with ADSI objects

You've actually already seen a quick example of working with Active Directory objects when we set the **Description** property of our test user account back in the "Using ADSI Objects" section. Here's how to change a password.

```
$user.SetPassword("P@ssw0rd!")
```

Retrieve the object into a variable, and then call its **SetPassword()** method, passing the desired new password as an argument.

Creating new objects is straightforward: you'll need to retrieve the parent container that you want the new object created in, and then call the parent's **Create()** method. Doing so will return an object that represents the new directory object; you'll need to set any mandatory properties, and then save the object to the directory.

```
[ADSI]$container = "LDAP://ou=employees,dc=company,dc=local"
$user = $container.Create("user", "cn=Tim E. Clock")
$user.put("SAMAccountName", "tclock")
$user.SetInfo()
```

If you're familiar with VBScript, you may be thinking: "Wow! This looks a lot like what we did in VBScript." It sure does—it's almost the same, in fact.

We should show you one more thing before we go on. Some AD properties (like WMI properties) are arrays, meaning they contain multiple values. For example, the **Member** property of an AD group object contains an array of **DistinguishedNames**, with each representing one group member. Here's an example of retrieving the **DistinguishedName** of the first member of a domain's Domain Admins group.

```
[ADSI]$group = "LDAP://cn=Domain Admins,cn=Users,dc=company,dc=local"
$group.member[0]
```

Modifying these properties is a bit complicated, since you have to use the **PutEx()** method and pass it a parameter indicating whether you're clearing the property completely, updating a value, adding a value, or deleting a value. The special parameter values are:

- Clear the property.
- Change an existing value within the property.
- Add a new value to the property.
- Delete a value from the property.

So, here's how you would add a new user to the Domain Admins group, which we've already retrieved into the `$group` variable.

```
$group.PutEx(3, "member", @("cn=Jason Helmick,ou=Employees,dc=company,dc=local"))
$group.SetInfo()
```

We used the value 3, so we're adding a value to the array. We have to actually add an array, even though we only need to have one item in the array—the user we want to add to the group. So, we use PowerShell's `@` operator to create a new, one-element array containing the **DistinguishedName** of the new group member. You need to use **SetInfo()** to save the information back to the directory.

We hope this quick overview gives you a good start in using ADSI from within PowerShell. As you can see, the actual mechanics of it all aren't that complicated. The tough part is understanding what's going on inside the directory, including the property names that let you view and modify the information you need.

## In Depth 10

# Managing Active Directory with ADSI, Part 2

## Managing Active Directory with ADSI

As we described in Chapter 9, Windows PowerShell's current support for ADSI is pretty limited. The Active Directory (AD) team at Microsoft has released a module of cmdlets for AD that work back to Windows 2003. Full coverage of those cmdlets is outside the scope of this book, although we will give you a small sampling to whet your appetite. We also encourage you to look at SAPIEN's TFM "Managing Active Directory with Windows PowerShell" by Jeffery Hicks, for a complete rundown. This chapter focuses on managing directory services by using the .NET Framework **DirectoryService** classes, which you can access by using the **New-Object** cmdlet.

```
$Root = New-Object DirectoryServices.DirectoryEntry "LDAP://DC=MyCompany,dc=local"
```

However, the PowerShell team realized most admins won't have experience programming in .NET Framework, so they developed an **[ADSI]** type adapter, which we started with in the In Depth Chapter 9. This type adapter abstracts the underlying .NET Framework classes and makes it a little bit easier to work with AD objects.

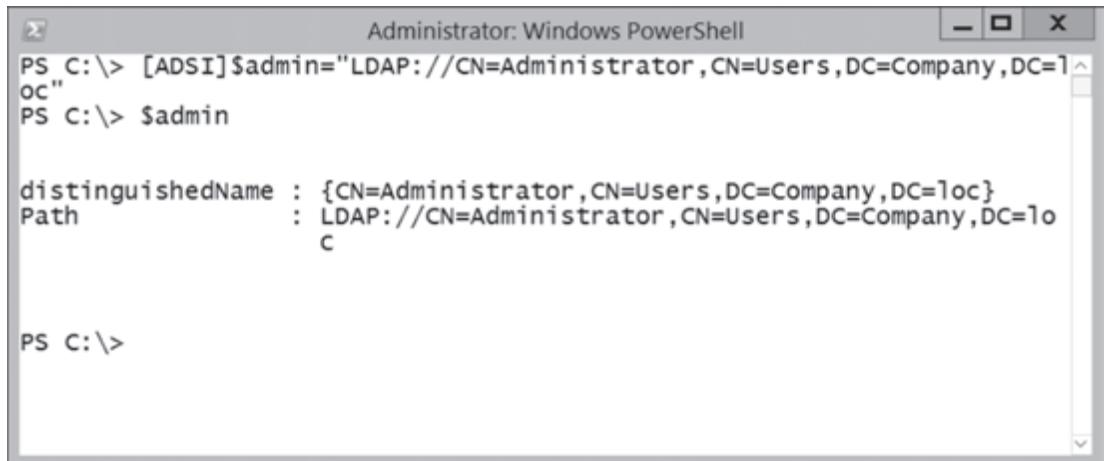
As you work with the **[ADSI]** type adapter, you'll realize there are limitations. Wouldn't it be nice to mount your AD store like any other file system? PowerShell itself does not ship with a directory services provider, but the free PowerShell Community Extensions includes one. The AD cmdlets shipping with Windows Server 2008 R2 also include a PSDrive provider. When installed, the provider will create a new PSDrive that is mapped to the root of your AD domain. You can then navigate AD just like any other drive.

Depending on your needs, there are several approaches to working with AD, without resorting to third-party extensions. We'll show you several example tasks in this chapter.

## Working with users by using the [ADSI] type adapter

The PowerShell team introduced the [ADSI] type adapter in PowerShell version 1. This type adapter makes it easier to create, modify, display, and delete AD objects such as users, groups, and computers. In order to use the type adapter, you need to know the distinguished name of the object.

```
[ADSI]$admin = "LDAP://CN=Administrator,CN=Users,DC=Company,DC=loc"  
$admin
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The window shows the following command and its output:

```
PS C:\> [ADSI]$admin="LDAP://CN=Administrator,CN=Users,DC=Company,DC=loc"  
PS C:\> $admin  
  
distinguishedName : {CN=Administrator,CN=Users,DC=Company,DC=loc}  
Path : LDAP://CN=Administrator,CN=Users,DC=Company,DC=loc  
C  
  
PS C:\>
```

Figure 10-1

You can retrieve a great amount of information from the object, such as group information.

```
$admin.memberOf  
$admin.whenChanged
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$admin.memberOf is run, listing several Active Directory groups: CN=Group Policy Creator Owners, CN=Users, DC=Company, DC=loc, CN=Domain Admins, CN=Users, DC=Company, DC=loc, CN=Enterprise Admins, CN=Users, DC=Company, DC=loc, CN=Schema Admins, CN=Users, DC=Company, DC=loc, and CN=Administrators, CN=BuiltIn, DC=Company, DC=loc. Below this, the command PS C:\> \$admin.whenChanged is run, followed by the date and time: wednesday, July 10, 2013 10:46:06 PM. Finally, PS C:\> is shown at the bottom.

```
PS C:\> $admin.memberOf
CN=Group Policy Creator Owners,CN=Users,DC=Company,DC=loc
CN=Domain Admins,CN=Users,DC=Company,DC=loc
CN=Enterprise Admins,CN=Users,DC=Company,DC=loc
CN=Schema Admins,CN=Users,DC=Company,DC=loc
CN=Administrators,CN=BuiltIn,DC=Company,DC=loc
PS C:\> $admin.whenChanged

wednesday, July 10, 2013 10:46:06 PM

PS C:\>
```

Figure 10-2

Piping the **\$admin** variable to the **Get-Member** cmdlet will list what appears to be all the available ADSI properties of the object. If you are familiar with ADSI, you'll realize that some properties are missing. The **Get-Member** cmdlet only displays properties with defined values. You can modify other properties as long as you already know the property name, which we'll show you later. One other important reminder is that when you create an object, PowerShell stores the property values in a local cache. If a user modifies the object in AD, you won't see the changes locally unless you refresh the cache. Here's how we would do it with the example in Figure 10-2.

```
$admin.RefreshCache()
```

We can also use the **[ADSI]** type accelerator to create an object in AD. Let's take a look at the **CreateUser.ps1** script.

### CreateUser.ps1

```
#specify the OU where you want to create the account
[ADSI]$OU = "LDAP://OU=Employees,DC=MyCompany,DC=Local"

#Add the user object as a child to the OU
$newUser = $OU.Create("user", "CN=Jack Frost")
$newUser.Put("SAMAccountName", "jfrost")

#commit changes to Active Directory
$newUser.SetInfo()

#set a password
$newUser.SetPassword("P@ssw0rd")

#define some other user properties
$newUser.Put("DisplayName", "Jack Frost")
$newUser.Put("UserPrincipalName", "jfrost@mycompany.com")
$newUser.Put("GivenName", "Jack")
```

```
$newUser.Put("sn", "Frost")

#enable account = 544
#disable account = 546
$newUser.Put("UserAccountControl", "544")

$newUser.Put("Description", "Created by PowerShell $((get-date).ToString())")

#commit changes to Active Directory
$newUser.SetInfo()

#flag the account to force password change at next logon
$newUser.Put("pwdLastSet", 0)
$newUser.SetInfo()
```

Before you can create an object, you first must create an object for the parent container. In this example, we're using the Employees organizational unit. To create a new user object, we simply invoke the parent object's **Create()** method and specify the type of child object and its name.

```
$newUser = $OU.Create("user", "CN=Jack Frost")
```

To define properties, we'll use the **Put()** method. When you create a user account, you have to also define the **SAMAccountName**.

```
$newUser.Put("SAMAccountName", "jfrost")
```

Before we can set any other properties, you need to write the object from the local cache to AD. You can accomplish this by calling the **SetInfo()** method.

```
$newUser.SetInfo()
```

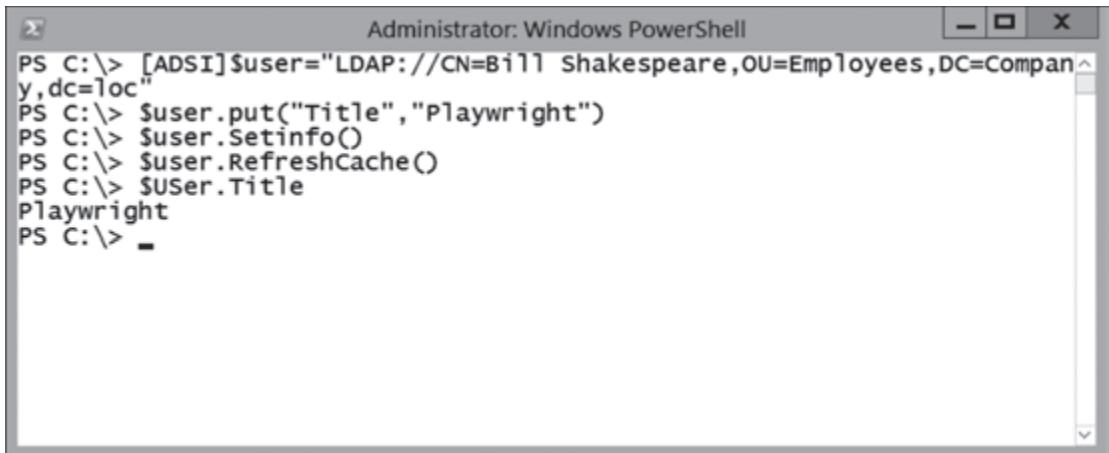
To set the user's password, there is a **SetPassword()** method that takes the new password as a parameter.

```
$newUser.SetPassword("P@ssw0rd")
```

Once this is accomplished, we can define some additional properties, by using the **Put()** method, as you can see in the remainder of the script.

To modify an existing user, it is merely a matter of creating an ADSI user object and using the **Put()** method to define user attributes. Don't forget to call the **SetInfo()** method or none of your changes will be committed to AD.

```
[ADSI]$user = "LDAP://CN=Bill Shakespeare,OU=Employees,DC=Company,dc=loc"
$user.put("Title", "Playwright")
$user.Setinfo()
$user.RefreshCache()
$user.Title
```

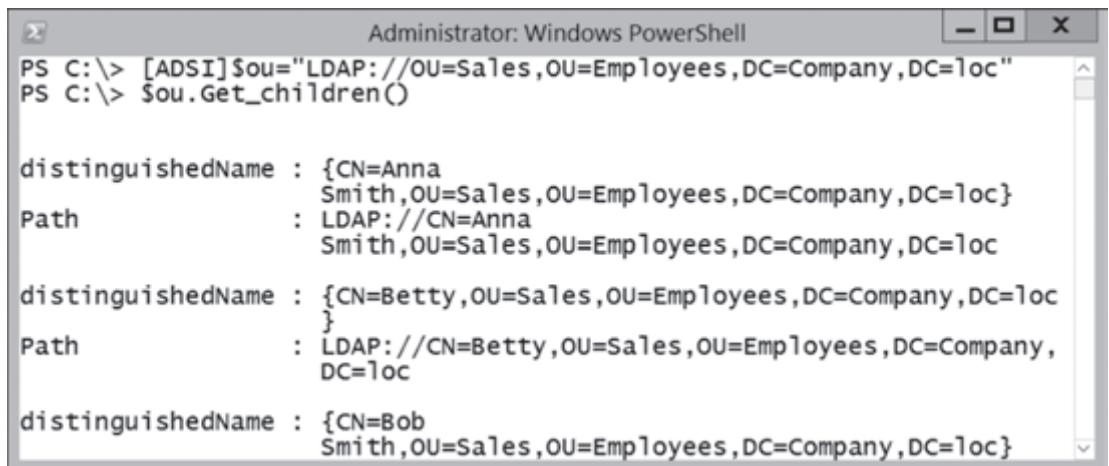


```
Administrator: Windows PowerShell
PS C:\> [ADSI]$user="LDAP://CN=Bill Shakespeare,OU=Employees,DC=Company,DC=loc"
PS C:\> $user.put("Title","Playwright")
PS C:\> $user.Setinfo()
PS C:\> $user.RefreshCache()
PS C:\> $user.Title
Playwright
PS C:\> _
```

Figure 10-3

To delete a user, first we create an object for the parent container, typically an organizational unit.

```
[ADSI]$ou = "LDAP://OU=Sales,OU=Employees,DC=MyCompany,DC=loc"
$ou.Get_Children()
```



```
Administrator: Windows PowerShell
PS C:\> [ADSI]$ou="LDAP://OU=Sales,OU=Employees,DC=Company,DC=loc"
PS C:\> $ou.Get_children()

distinguishedName : {CN=Anna
Smith,OU=Sales,OU=Employees,DC=Company,DC=loc}
Path              : LDAP://CN=Anna
Smith,OU=Sales,OU=Employees,DC=Company,DC=loc

distinguishedName : {CN=Betty,OU=Sales,OU=Employees,DC=Company,DC=loc
}
Path              : LDAP://CN=Betty,OU=Sales,OU=Employees,DC=Company,
DC=loc

distinguishedName : {CN=Bob
Smith,OU=Sales,OU=Employees,DC=Company,DC=loc}
```

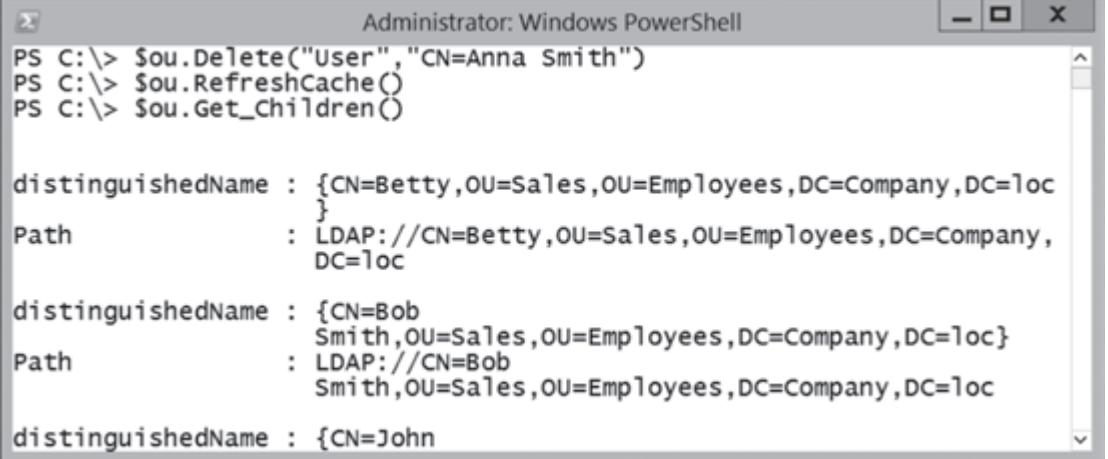
Figure 10-4

Then we use the **Delete()** method.

```
$ou.Delete("User", "CN=Anna Smith")
```

Refresh the cache and you'll notice that the user is no longer in AD.

```
$ou.RefreshCache()
$ou.Get_Children()
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$ou.Delete("User", "CN=Anna Smith") is run, followed by PS C:\> \$ou.RefreshCache() and PS C:\> \$ou.Get\_Children(). The output shows the deletion of three user objects:

```
distinguishedName : {CN=Betty,OU=Sales,OU=Employees,DC=Company,DC=loc}
Path              : LDAP://CN=Betty,OU=Sales,OU=Employees,DC=Company,
distinguishedName : {CN=Bob
                     Smith,OU=Sales,OU=Employees,DC=Company,DC=loc}
Path              : LDAP://CN=Bob
                     Smith,OU=Sales,OU=Employees,DC=Company,DC=loc
distinguishedName : {CN=John
```

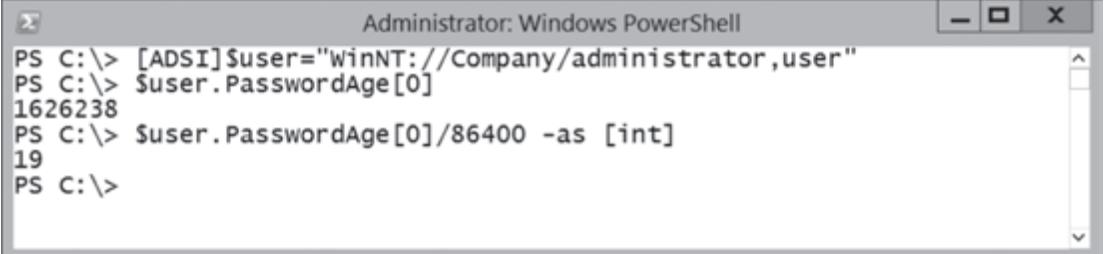
Figure 10-5

There is no need in this situation to call the **SetInfo()** method. As soon as you invoke the **Delete()** method, the object is gone. You can use this method to delete any object. All you have to do is specify the object class and its name.

## Obtaining password age

The easiest way to obtain the password age for a user or a computer is to use the WinNT provider and look at the **PasswordAge** property.

```
[ADSI]$user = "WinNT://Company/administrator,user"
$user.PasswordAge[0]
$user.PasswordAge[0]/86400 -as [int]
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> [ADSI]\$user = "WinNT://Company/administrator,user" is run, followed by PS C:\> \$user.PasswordAge[0]. The output shows the password age of 1626238, which is then converted to days using PS C:\> \$user.PasswordAge[0]/86400 -as [int], resulting in 19.

```
PS C:\> [ADSI]$user = "WinNT://Company/administrator,user"
PS C:\> $user.PasswordAge[0]
1626238
PS C:\> $user.PasswordAge[0]/86400 -as [int]
19
PS C:\>
```

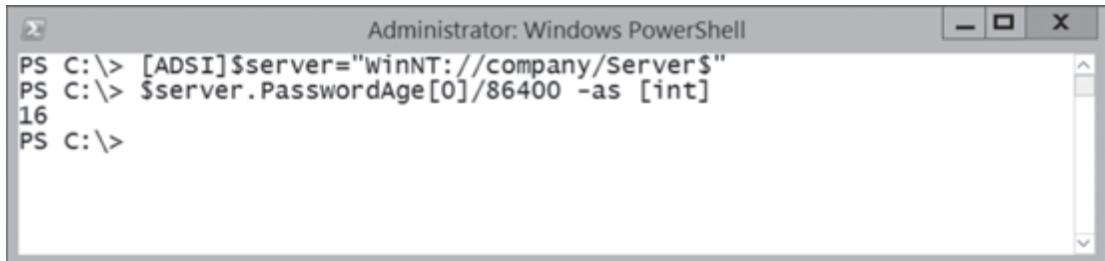
Figure 10-6

The first step is to create an ADSI object for the user, by employing the WinNT provider. In this example, we are getting the user object for Administrator in the Company domain.

PowerShell stores the password age in the **Password** property, but returns it as a single element array. Therefore, we reference it with an index number of 0. The value is in seconds, so we divide it by 86400 to obtain the number of days and cast the result as an integer, which in effect rounds the value. As you can see, the admin's password was last changed 19 days ago.

You can use the password age for a computer account to identify obsolete computer accounts. If the password has not changed in, say, 45 days, it is very likely the computer account is no longer active.

```
[ADSI]$server = "WinNT://company/Server$"
$server.PasswordAge[0]/86400 -as [int]
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> [ADSI]\$server="WinNT://company/Server\$" is entered, followed by PS C:\> \$server.PasswordAge[0]/86400 -as [int]. The output shows the result as 16, indicating the password age in days. The window has standard operating system window controls (minimize, maximize, close) at the top right.

Figure 10-7

The only difference with this code, compared to the code for a user account, is that we must specify the **SAMAccountName** of the computer, which should be the computer's NetBIOS name, appended with the \$ sign. The remaining code is the same. In this example, it is very clear that server **Server\$** password is not obsolete since its password age is only 16 days old.

## Deleting users

Deleting a user is a very straightforward task. All you need is the distinguished name of the container or organizational unit and the Active Directory (AD) name of the user object.

```
[ADSI]$ou = "LDAP://OU=employees,DC=MyCompany,dc=local"
$ou.Delete("user", "CN=Sam Hamm")
```

The first line creates an ADSI object that represents the parent container—in this case, the Employees OU. The second line calls the **Delete()** method, which requires the type of object and its canonical name.

## Bulk-creating users

With a little extra effort, we can expand the previous example and create a group of users in bulk. We feel that today you're better off using the AD cmdlets, rather than ADSI, for this task, but you may have older scripts to maintain that use this similar idea. The following script will create a group of users based on information stored in a comma-separated value (CSV) file.

### Import Users.ps1

```
#Import-Users.ps1

$data = "newusers.csv"
$imported = Import-Csv $data

#retrieve list of csv column headings
#Each column heading should correspond to an ADSI user property name

$properties = $imported |Get-Member -type noteProperty |
where { $_.name -ne "OU" -and $_.name -ne "Password" ` 
-and $_.name -ne "Name" -and $_.name -ne "SAMAccountName" }

for ($i = 0;$i -lt $imported.count;$i++) {
    Write-Host "Creating User"$imported[$i].Name "in" $imported[$i].OU

    [ADSI]$OU = "LDAP://"+$imported[$i].OU

    $newUser = $OU.Create("user", "CN="+$imported[$i].Name)
    $newUser.Put("SAMAccountName", $imported[$i].SAMAccountName)
    #commit changes to Active Directory
    $newUser.SetInfo()
    #set a password
    $newUser.SetPassword($imported[$i].Password)
    $newUser.SetInfo()

    foreach ($prop in $properties) {
        #set additional properties
        $value = $imported[$i].($prop.name)
        if ($value.length -gt 0) {
            #only set properties that have values
            $newUser.put($prop.name, $value)
        }
    }
    $newUser.SetInfo()
}
```

Our script assumes the CSV file will have the required column headings of **OU**, **Name**, **SAMAccountName**, and **Password**. The **OU** column will contain the distinguished name of the organizational unit, where PowerShell will create the new user account, such as OU=Employees, DC=MyCompany, DC=Local. The **Name** property will be the user's AD name and the **SAMAccountName** property will be the user's down-level logon name. You can have as many other entries as you want. Each column heading must correspond to an ADSI property name. For example, use **SN** for the user's last name and **GivenName** for the user's first name. The script begins by using the **Import-CSV** cmdlet to import the CSV file.

```
$data = "newusers.csv"
$imported = Import-Csv $data
```

Since the CSV file will likely contain column headings for additional user properties, we'll create an object to store those property names. We'll exclude the required columns by selecting all property names that don't match the required names.

```
$properties = $imported | Get-Member -type noteProperty | ` 
where { $_.name -ne "OU" -and $_.name -ne "Password" ` 
-and $_.name -ne "Name" -and $_.name -ne "SAMAccountName" }
```

Armed with this information, we can now run through the list of new user information by using the **For** construct.

```
for ($i = 0; $i -lt $imported.count; $i++) {
    Write-Host "Creating User" $imported[$i].Name "in" $imported[$i].OU
```

The **\$imported** variable is an array, so we can access each array member by using the array index. We'll first create an object for the OU where the user will be created, by using the **[ADSI]** type adapter.

```
[ADSI]$OU = "LDAP://" +$imported[$i].OU
```

Now we can create the new user by referencing the required imported user properties.

```
$newUser = $OU.Create("user", "CN=" +$imported[$i].Name)
$newUser.Put("SAMAccountName", $imported[$i].SAMAccountName)
#commit changes to Active Directory
$newUser.SetInfo()
```

At this point, we can add any other user properties that are defined in the CSV file. We accomplish this by enumerating the property list object.

```
foreach ($prop in $properties) {
```

For each property name, we'll retrieve the corresponding value from the current user.

```
$value = $imported[$i].($prop.name)
```

For example, if the property name is **Title**, then the script sets the **\$value** variable to the value of **\$imported[\$i].Title**. We put the **\$prop.name** variable in parentheses to instruct PowerShell to evaluate the expression, so it will return **Title** in this example.

As written, this script can only set single-valued properties that accept strings. The script checks the length of the **\$value** variable. A length of 0 means that there is no value and there's no reason to attempt to set the property. So, if the length of **\$value** is greater than 0, we know there is a value, and we'll set the user property with it.

```
if ($value.length -gt 0) {
    #only set properties that have values
    $newUser.put($prop.name, $value)
}
```

After you have set all the properties, we call the **SetInfo()** method to write the new information to Active Directory.

```
$newUser.SetInfo()
```

We repeat this process for every user imported from the CSV file.

## Working with computer accounts

Creating a new computer account is very similar to creating a new user account and, in many ways it is much easier because there are very few properties that you have to define. Here's a function that you can use to create a new computer account.

### CreateNewComputer.ps1

```
Function New-Computer {
    Param([string]$name = $(Throw "You must enter a computer name."),
        [string]$Path = "CN=computers,DC=MyCompany,DC=Local",
        [string]$description = "Company Server",
        [switch]$enabled)

    [ADSI]$OU = "LDAP://$Path"
    #set name to all uppercase
    $name = $name.ToUpper()

    $computer = $OU.Create("computer", "CN=$name")
    $computer.Put("SAMAccountName", $name)
    $computer.Put("Description", $description)

    if ($enabled) {
        $computer.Put("UserAccountControl", 544)
    } else {
        $computer.Put("UserAccountControl", 546)
    }

    $computer.SetInfo()
} #end function
```

The function requires the name of the new computer object and, optionally, the organizational unit path, a description, and whether the account should be enabled. It is disabled by default, unless you use the **-Enabled** parameter. Default values are specified for the optional parameters.

The function creates an ADSI object for the OU or container where you want to create the computer object.

```
[ADSI]$OU = "LDAP://$Path"
```

The function next creates the computer object in the container specifying the Active Directory name and the **SAMAccountName**.

```
$computer = $OU.Create("computer", "CN=$name")
$computer.Put("SAMAccountName", $name)
The function defines the description:
$computer.Put("Description", $description)
```

By default, PowerShell disables the computer account, but you can specify to create the accounts as enabled. The **UserAccountControl** property defines this setting.

```
if ($enabled) {
    $computer.Put("UserAccountControl", 544)
} else {
    $computer.Put("UserAccountControl", 546)
}
```

Finally, we call the **SetInfo()** method to write the new account to the Active Directory database.

```
$computer.SetInfo()
```

## Delete computer accounts

Deleting a computer account is essentially the same as deleting user accounts. All you need are the distinguished name of the container or organizational unit and the Active Directory name of the computer object.

```
[ADSI]$ou = "LDAP://OU=Desktops,DC=Company,dc=loc"
$ou.Delete("computer", "CN=Comp1")
```

The first line creates an ADSI object, which represents the parent container—in this case, the Desktops OU. The second line calls the **Delete()** method, which requires the type of object and its canonical name.

## Working with groups

Creating a group is very similar to creating a user.

```
[ADSI]$OU = "LDAP://OU=Employees,DC=Company,dc=loc"
$NewGroup = $OU.Create("group", "CN=Marketing")
$NewGroup.Put("SAMAccountName", "Marketing")
$NewGroup.Put("Description", "Marketing People")
$NewGroup.SetInfo()
```

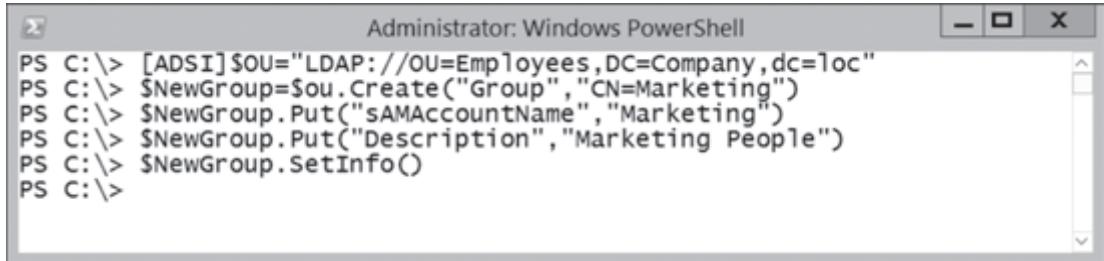


Figure 10-8

If you are familiar with modifying group memberships in ADSI, it's not too different conceptually in PowerShell. As with ADSI, you need a **DirectoryEntry** object for the group. You also need to know the distinguished name of the user object that you want to add. Armed with that information, it's a matter of adding the user's distinguished name to the object's **Member** property. Here's a script that demonstrates how to modify group membership.

### AddToGroup.ps1

```
#AddToGroup.ps1

[ADSI]$Grp = "LDAP://CN=Marketing,OU=Employees,DC=Company,DC=loc"
$NewUserDN = "CN=Bob Smith,OU=Employees,DC=Company,DC=loc"

#create an array object from current group members
$grpMembers = @($Grp.Member)

#display current group membership
Write-Host "There are currently $($grpMembers.Count) members in $($Grp.Name)" -ForegroundColor cyan
foreach ($user in $grpMembers) { $user }

Write-Host `n; Write-Host "Adding $NewUserDN" -ForegroundColor cyan
($Grp.Member).add($NewUserDN) > $NULL

#commit changes to Active Directory
$Grp.SetInfo()

#refresh object and display new membership list
$Grp.refreshCache()
```

```
$grpMembers = @($grp.Member)

#display new membership
Write-Host "There are now $($grpMembers.Count) members in $($grp.Name)" -ForegroundColor cyan
foreach ($user in $grpMembers) {
    if ($user -eq $NewUserDN) {
        Write-Host $user -ForegroundColor Green
    }
    else
    {
        write-Host $user -ForegroundColor Yellow
    }
}
```

This script creates an ADSI object for the Marketing group and also creates an object for the current membership list that is displayed, by using the **Foreach** loop. Adding the new user appears a little confusing at first.

```
($grp.Member).Add($NewUserDN) > $NULL
```

What we need to do is to call the **Add()** method for the group's **Member** property, which is a collection, and then specify the user's distinguished name. By the way, if we wanted to nest another group, we would specify that group's distinguished name. The reason we redirect output to the **\$Null** variable is purely cosmetic. Without the redirection, the expression returns the number of members currently in the group. In the course of running the script, displaying that number here serves no purpose and is distracting. We eliminate it by redirecting any output to **\$Null**.

None of this work means anything until we commit the change to Active Directory, by using the **SetInfo()** method. The script finishes by refreshing the local cache and listing its new members, indicating the new users in a green font and existing users in yellow.

## Moving objects

PowerShell and the .NET Framework use a slightly different method for moving objects in Active Directory. You should be able to call the **MoveTo()** method.

```
[ADSI]$obj = "LDAP://CN=Comp2,CN=Computers,DC=Company,dc=loc"
$obj.MoveTo("LDAP://OU=Desktops,DC=Company,dc=loc")
```

## Searching for users

We'll wrap up this chapter by showing you how easy it is to search in Active Directory with PowerShell. Since PowerShell is based on the .NET Framework, it can create a **DirectorySearcher** object by using the **[ADSISeacher]** type adapter. Here's a short script that will return the distinguished name of every user account in an Active Directory domain.

## SearchForAllUsers.ps1

```
#SearchForAllusers.ps1
[ADSISearcher]$searcher = "(&(objectcategory = person)(objectclass = user))"

#enable paged searching
$searcher.pagesize = 50
$users = $searcher.FindAll()

#display the number of users
Write-Host "There are $($users.count) users in this domain." -Foregroundcolor cyan
#display each user's distinguishedname
foreach ($user in $users) {
    Write-Host $user.properties.distinguishedname -Foregroundcolor green
}
```

We use the **[ADSISearcher]** type adapter to create a **DirectorySearcher** object. It is just as easy to specify the filter when creating the object. The filter is an LDAP query string. In this case, we want to find all objects that are basically user accounts. The **DirectorySearcher** object has two methods that you are most likely to use: **FindAll()** and **FindOne()**. The former will return all objects that match the query and the latter will only return the first one it finds. In this script, we create a new object to hold the query results. We can then use the **Foreach** construct to display the **DistinguishedName** property of each user in the result collection.

## Fun with LDAP Filters

You don't have to have extensive knowledge about LDAP to build a complex query. If you are running Windows 2003 or later, you already have a tool that will do it for you. In Active Directory Users and Computers, there is a **Saved Queries** feature. When you create a query, Active Directory creates an LDAP query string. All you need to do is copy the string and use it as the directory searcher filter. For example, we created a saved query to find all disabled users that created the following LDAP query string.

```
(&(objectCategory = person) (objectClass = user) (userAccountControl:1.2.840.113556.1.4.803: = 2)).
```

When we substitute this string for the filter in SearchForAllUsers.ps1, we receive a list of every disabled user. The tool is pretty powerful and can create some very complex query strings. Now, you can also use them in your PowerShell scripts.

There is a subtle but important fact to remember when using the **DirectorySearcher** object. The objects returned by the query aren't really the Active Directory objects, but are more like pointers. The search result can give you some property information like **DistinguishedName**, but if you want more specific object information, you need to get the object itself. The following script is slightly modified from SearchForAllUsers.ps1.

**SearchForAllUsersAdvanced.ps1**

```
#SearchForAllUsersAdvanced.ps1
[ADSI searcher] $searcher = "(&(objectcategory = person)(objectclass = user))"
$searcher.pagesize = 50
$users = $searcher.FindAll()

#display the number of users
Write-Host "There are $($users.count) users in this domain." -Foregroundcolor cyan

#display user properties
foreach ($user in $users) {
    foreach ($user in $users) {
        $entry = $user.GetDirectoryEntry()
        $entry | Select displayname, SAMAccountName, description, distinguishedname
    }
}
```

In the **Foreach** loop, we create a new variable called **\$entry**, by invoking the **GetDirectoryEntry()** method of the object that the query returned. This object gives us access to all the properties in Active Directory. In this script, we selected to show **DisplayName**, **SAMAccountName**, **Description**, and **DistinguishedName**.

We mentioned that the **DirectorySearcher** object also has a **FindOne()** method. This is very useful when you know there is only one result, such as finding the **DistinguishedName** of a user when all you know is the user's **SAMAccountName**.

**FindUserDN.ps1**

```
#FindUserDN.ps1
$sam = Read-Host "What user account do you want to find?"
[ADSI searcher] $searcher = "(&(objectcategory = person)(objectclass = user)(SAMAccountName = $sam))"

$searcher.pagesize = 50
$results = $searcher.FindOne()

if ($results.path.length -gt 1)
{
    write $results.path
}
else
{
    Write-Warning "User $sam was not found."
}
```

This script is very similar to the other searching scripts. The primary difference is that the search filter is looking for user objects where the **SAMAccountName** is equal to a value specified by the user, by using the **Read-Host** cmdlet. Since we know there will only be one result, the searcher can stop as soon as it finds a match. We've added some error checking in the script. If the script finds a match, then the length of the path property will be greater than 1 and we can display it. Otherwise, there is no path, which means the script found no match and we can display a message to that effect.

The **[ADSI]** and **[ADSISeacher]** type adapters are merely wrappers for .NET Framework directory service objects. They abstract much of the underlying functionality, making them easy to use. You can create PowerShell scripts that directly create and manipulate the .NET Framework objects, but in our opinion, that process is more like systems programming than scripting.

## Microsoft Active Directory cmdlets

With the release of Windows Server 2008 R2, Microsoft released a PowerShell solution for managing Active Directory. Here is a brief look—remember: Jeff Hicks wrote an entire TFM on this topic, so if you need more information, check it out.

First, we need to load the Active Directory cmdlets. If you’re running PowerShell version 2 (and you shouldn’t be), the first step is to import the module. Where do you find the module? You can do this by using implicit remoting (discussed in the Learning Guide) or installing the Remote Server Administration Tools (RSAT) locally. (We have them installed locally for this.)

You do not need to import the module in PowerShell version 3 and version 4, because the module will load dynamically the first time you try to use a cmdlet. This is one of the primary reasons that if you’re still using version 2, you need to upgrade. After all—it’s free.

```
Import-Module ActiveDirectory
```

You can retrieve a list of the cmdlets, by using **Get-Command** and the **-Module** parameter.

```
get-command -Module ActiveDirectory | Select name
```

```
Administrator: Windows PowerShell
PS C:\> get-command -Module ActiveDirectory | Select name
Name
-----
Add-ADCentralAccessPolicyMember
Add-ADComputerServiceAccount
Add-ADDomainControllerPasswordReplicationPolicy
Add-ADFineGrainedPasswordPolicySubject
Add-ADGroupMember
Add-ADPrincipalGroupMembership
Add-ADResourcePropertyListMember
Clear-ADAccountExpiration
Clear-ADClaimTransformLink
Disable-ADAccount
Disable-ADOptionalFeature
Enable-ADAccount
```

Figure 10-9

How about creating a new user account? With the cmdlets, we will use the **New-ADUser** cmdlet. The cmdlet requires that you pass the password as a secure string, so we'll first get one by using the **Read-Host** cmdlet.

```
$userpwd = Read-host "Enter default password" -AsSecureString
```

We can use this variable in an expression.

```
New-ADUser -Name "Huck Finn" -AccountPassword $userpwd ` 
-ChangePasswordAtLogon $True -Enabled $True -GivenName "Huck" -Surname "Finn" ` 
-SAMAccountName "HuckF"
```

```
Administrator: Windows PowerShell
PS C:\> $userpwd=Read-host "Enter default password" -AsSecureString
Enter default password: *****
PS C:\> New-ADUser -Name "Huck Finn" -AccountPassword $userpwd -Change
>PasswordAtLogon $True -Enabled $True -GivenName "Huck" -Surname "Finn"
-SamAccountName "HuckF"
PS C:\> Get-ADUser -Identity HuckF

DistinguishedName : CN=Huck Finn,CN=Users,DC=Company,DC=loc
Enabled          : True
GivenName        : Huck
Name             : Huck Finn
ObjectClass      : user
ObjectGUID       : c5115a24-bdea-48bd-8bd7-cd4d1f1f355e
SamAccountName   : HuckF
SID              : S-1-5-21-1543093795-411134941-608708178-1615
```

Figure 10-10

If you don't specify a distinguished name for the **Path** property, PowerShell will create the account in the default **USERS** container. By default, the cmdlet doesn't write anything to the pipeline, so we could use the **-PassThru** parameter to see the results. This is the same information we would see if we had used the **Get-ADUser** cmdlet.

```
Get-ADUser -Identity Huckf
```

Oops. We forgot a few things, so let's modify the account by using the **Set-ADUser** cmdlet.

```
Get-ADUser -Identity Huckf | Set-ADUser -Title "Lab Technician" ` 
-DisplayName "Huckleberry Finn"
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Get-ADUser -Identity HuckF | Set-ADUser -Title "Lab Technician" "-DisplayName "Huckleberry Finn"" is run. This is followed by PS C:\> Get-ADUser -Identity huckf -Properties \* | Select title, displayname. The output displays the user's title and display name.

title	displayname
Lab Technician	Huckleberry Finn

Figure 10-11

Notice that if you want to see specific properties, you have to ask for them by using the **-Properties** parameter.

```
Get-ADUser -Identity huckf -properties * | Select title, displayname
```

As you can see, using cmdlets is not that difficult. The most time-consuming part is studying the Help and examples for these new cmdlets.

## In Depth 11

# Using WMI to manage systems

## Using WMI to manage systems

Using WMI in the console and administrative scripts is a common practice. WMI is an extremely powerful way to manage just about every aspect of a system, including hardware, software, and configuration. You can also use it to remotely manage systems.

PowerShell has the two types of cmdlets that help you work with WMI. We know that's confusing, but starting in PowerShell version 3, a new and improved set was added.

## Why all these cmdlets

The older PowerShell cmdlets can be listed by using **Get-Help**.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> get-help \*wmi\*". The output is a table listing cmdlets related to WMI:

Name	Category	Module
gwmi	Alias	
iwmi	Alias	
rwmi	Alias	
swmi	Alias	
Get-WmiObject	Cmdlet	Microsoft.PowerShell.M...
Invoke-WmiMethod	Cmdlet	Microsoft.PowerShell.M...
Register-WmiEvent	Cmdlet	Microsoft.PowerShell.M...
Remove-WmiObject	Cmdlet	Microsoft.PowerShell.M...
Set-WmiInstance	Cmdlet	Microsoft.PowerShell.M...
about_WMI	HelpFile	
about_Wmi_Cmdlets	HelpFile	

Figure 11-1

These cmdlets use DCOM (RPC) to access remote computers. DCOM is problematic—it can be slow and very difficult to use against remote computers that are across firewalls.

Starting with PowerShell version 3, a new cmdlet set was added that can use either DCOM or PowerShell Remoting (much faster), and conforms more to the CIM standard.

```
$Dcom = New-CimSessionOption -Protocol Dcom
$cimsession = New-CimSession -ComputerName RemoteComputerName -SessionOption $Dcom
Get-CimInstance -CimSession $cimsession -ClassName Win32_Share
Remove-CimSession $cimsession
```

To retrieve a list of these cmdlets, use **Get-Help \*CIM\***.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-Help \*cim\*". The output is a table listing cmdlets related to CIM:

Name	Category	Module
Get-CimAssociatedInstance	Cmdlet	CimCmdlets
Get-CimClass	Cmdlet	CimCmdlets
Get-CimInstance	Cmdlet	CimCmdlets
Get-CimSession	Cmdlet	CimCmdlets
Invoke-CimMethod	Cmdlet	CimCmdlets
New-CimInstance	Cmdlet	CimCmdlets
New-CimSession	Cmdlet	CimCmdlets
New-CimSessionOption	Cmdlet	CimCmdlets
Register-CimIndicationEvent	Cmdlet	CimCmdlets
Remove-CimInstance	Cmdlet	CimCmdlets
Remove-CimSession	Cmdlet	CimCmdlets
Set-CimInstance	Cmdlet	CimCmdlets

Figure 11-2

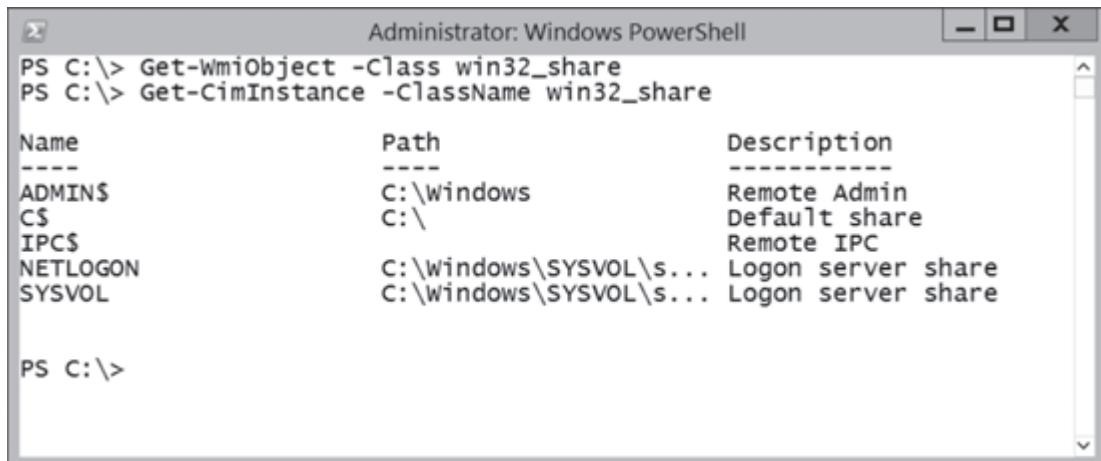
The important concept here is that you will see both used in this book and on the Internet. Realize that the CIM-based ones are newer and you should use those as you write automation scripts. However, because you will see both for a long time to come, you should be able to recognize and switch between the two. This chapter will use the older WMI and the newer CIM cmdlets interchangeably, so you can see the differences and become accustomed to working with both sets.

The **Get-WmiObject** (or **Get-CimInstance**) cmdlet acts as an interface to WMI. This cmdlet lets you access any WMI class in any WMI namespace. For our purposes, we'll stick to the default namespace of Root\cimv2 and the Win32 classes, since you'll use these classes in 95 percent of your scripts.

## Retrieving basic information

At its simplest, all you have to do is specify a class and run the **Get-WmiObject** cmdlet in order for the cmdlet to find all instances of that class and return a page of information.

```
Get-WmiObject -Class win32_share
Get-CimInstance -ClassName win32_share
```



```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject -Class win32_share
PS C:\> Get-CimInstance -ClassName win32_share
Name          Path          Description
----          ----          -----
ADMIN$        C:\Windows    Remote Admin
C$           C:\           Default share
IPC$          C:\Windows\SYSVOL\s... Logon server share
NETLOGON      C:\Windows\SYSVOL\s... Logon server share
SYSVOL       C:\Windows\SYSVOL\s... Logon server share

PS C:\>
```

Figure 11-3

In Figure 11-3, we asked PowerShell to display WMI information about all instances of the Win32\_Share class on the local computer. With this particular class, there are other properties that by default are not displayed. Depending on the class, you might receive different results. For example, the following expression displays a long list of properties and values.

```
Get-WmiObject win32_processor | Format-List *
```

```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject win32_processor | Format-List *

PSComputerName      : DC
Availability        : 3
CpuStatus           : 1
CurrentVoltage      : 33
DeviceID            : CPU0
ErrorCleared        :
ErrorDescription    :
LastErrorCode       :
LoadPercentage      : 0
Status              : OK
StatusInfo          : 3
AddressWidth        : 64
Datawidth           : 64
```

Figure 11-4

## Listing available classes

You're probably saying: "That's great, but how do I find out what Win32 classes are available?" All you have to do is ask. The **Get-WmiObject** cmdlet has a **-List** parameter that you can invoke.

```
Get-WmiObject -Class win32_* -List
Get-CimClass -ClassName win32_*
```

Notice that the **-Class** parameter allows you to use wildcard characters, much like the **Get-Help** cmdlet. The **Get-WmiObject** cmdlet requires that you use **-List**, or you will receive an error. The **Get-CimClass** cmdlet does not have a **-List** parameter. You can also pipe the list results to the **Where-Object** cmdlet and filter using PowerShell operators.

```
Get-WmiObject -List | where { $_.name -like "win32*" }
```

The **Get-WmiObject** cmdlet defaults to the root\CMIV2 namespace, but you can query available classes in any namespace.

```
Get-WmiObject -Namespace root\Security -List
Get-CimClass -Namespace root\security
```

CimClassName	CimClassMethods	CimClassProperties
__SystemClass	{}	{SECURITY...}
__thisNAMESPACE	{}	{provider}
__ProviderRegistration	{}	{provider...}
__EventProviderRegistration	{}	{provider...}
__ObjectProviderRegistration	{}	{provider...}
__ClassProviderRegistration	{}	{provider...}
__InstanceProviderRegistration	{}	{provider...}

Figure 11-5

All those classes starting with \_\_ are WMI system classes. You can't really hide them, but you're not really going to use them, either. You want the classes that don't start with \_\_.

## Listing properties of a class

The next question most of you are asking is: "How can I find out the properties for a given class?" As we discussed previously, you can use the **Get-Member** cmdlet to list all the properties for a WMI object. Take a look at the following script.

### List-WMIProperties.ps1

```
#List-WMIProperties.ps1
$class=Read-Host "Enter a Win32 WMI Class that you are interested in"

$wmi=Get-WmiObject -Class $class -Namespace "root\Cimv2"
$wmi | Get-Member -Membertype properties | where { $_.name -notlike "__*" } | Select name

Or, if using the CIM cmdlets, here is an example:
#List-WMIProperties.ps1
$class=Read-Host "Enter a Win32 WMI Class that you are interested in"

(Get-CimClass -ClassName $class -Namespace root/CIMV2 |
    select -Expand CimClassProperties | sort name).name
```

This script prompts you for a Win32 WMI class and defines a variable, by using the **Get-WmiObject** cmdlet for that class. We define a second variable that is the result of piping the first variable to the **Get-Member** cmdlet. Finally, we pipe the class object to the **Get-Member** cmdlet, by selecting only **Properties**. Because this will likely include system properties, we'll filter those out as well.

Usually, there are only a handful of properties in which you are interested. In this case, you can use the **-Property** parameter to specify which properties you want to display.

```
Get-WmiObject -Class win32_processor -Property name, caption, L2CacheSize
Get-CimInstance -ClassName Win32_Processor -Property name, caption, L2CacheSize
```

```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject -Class win32_processor -Property name, caption, L2CacheSize
PS C:\> Get-CimInstance -ClassName Win32_Processor -Property name, caption, L2CacheSize

Availability : :
CpuStatus   : :
CurrentVoltage : :
DeviceID    : :
ErrorCleared : :
ErrorDescription : :
LastErrorCode : :
LoadPercentage : :
Status       : CPU0
```

Figure 11-6

Here, we've asked for the **Win32\_Processor** class, and specifically the name, caption, and **L2CacheSize** properties. Unfortunately, the cmdlet also insists on displaying system properties, such as **\_Genus**. Since you don't care about those properties most of the time, here is a neater approach.

```
Get-WmiObject -Class win32_processor | Select-Object Name, Caption, L2CacheSize | Format-List
```

Here is another approach, using CIM.

```
Get-CimInstance -ClassName Win32_Processor -Property Name, Caption, L2CacheSize |
  Select-Object Name, Caption, L2CacheSize |
  Format-List
```

Here, we're calling the same cmdlet, except we use the **Select-Object** cmdlet to pick only the properties we're interested in seeing.

## Examining existing values

As you learn to work with WMI, it's helpful to look at what information is populated on a system. This is a great way to learn the different properties and what values you can expect. As you work with WMI more and more, you'll realize that not every property is always populated. There's no reason to spend time querying empty properties. With this in mind, we've put together a helpful script that enumerates all the properties for all the instances of a particular WMI class. However, the script only displays properties with a value and it won't display any of the system class properties like **\_Genus**.

**List-WMIVValues.ps1**

```
#List-WMIVValues.ps1
$class=Read-Host "Enter a Win32 WMI Class that you are interested in"
$wmiobjects = Get-CimInstance -ClassName $class -Namespace root/CIMV2
$properties = $wmiobjects | Get-Member -MemberType Property

Write-Host "Properties for $($class.ToUpper())" -ForegroundColor Yellow

foreach ($wmiobject in $wmiobjects) {
    foreach ($property in $properties) {
        if ($wmiobject.($property.name)) {
            Write-Host -ForegroundColor Green "$($property.name) = $($wmiobject.($property.name))"
        }
    }
    if ($wmiobjects.Count -gt 1) {
        Write-Host *****
    }
}
```

This script is based on our earlier **List-WMIProperties** script. Once we have the variable with all the properties, we iterate through all the instances of the specified WMI class. If the property value isn't null and the property name is not like `__*`, then the property name and its value are displayed. We've even thrown in a little color to spice up the display. You can also use an expression like the following:

```
Get-Wmiobject Win32_LogicalDisk | Select *
```

Or by using CIM.

```
Get-CimInstance -ClassName Win32_LogicalDisk | Select *
```

However, this will show you all properties, including system properties, regardless of whether they have a value.

## Getting information using SAPIEN's WMI Explorer

SAPIEN has a tool for retrieving namespaces, classes, property information, and useful Help regarding values—all in an easy-to-use GUI. This makes it exceptionally faster to find and locate the information you're interested in finding, and then adding it to a script.

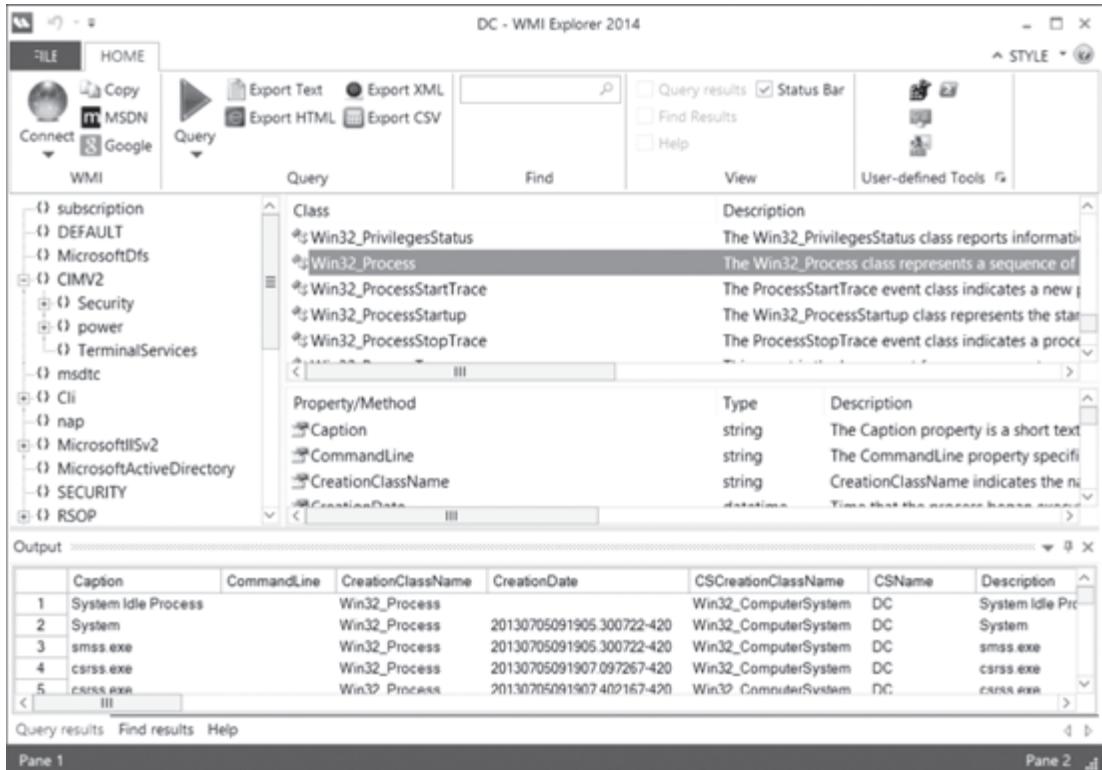


Figure 11-7

WMI Explorer makes it easy to connect to remote computers so that you can explore the namespaces and classes.

## Remote management

Now that you know about WMI classes and properties, we'll take a closer look at what you do with them. While you can make some system configuration changes with WMI, most of the WMI properties are read-only, which makes for terrific management reports. The **Get-WmiObject** cmdlet even has two parameters that make this easy to do on your network.

You can use the **-ComputerName** parameter to specify a remote computer that you want to connect to and the **-Credential** parameter to specify alternate credentials. However, you can't use alternate credentials when querying the local system.

```
Get-WmiObject win32_operatingsystem -ComputerName Server -Credential (Get-Credential)
```

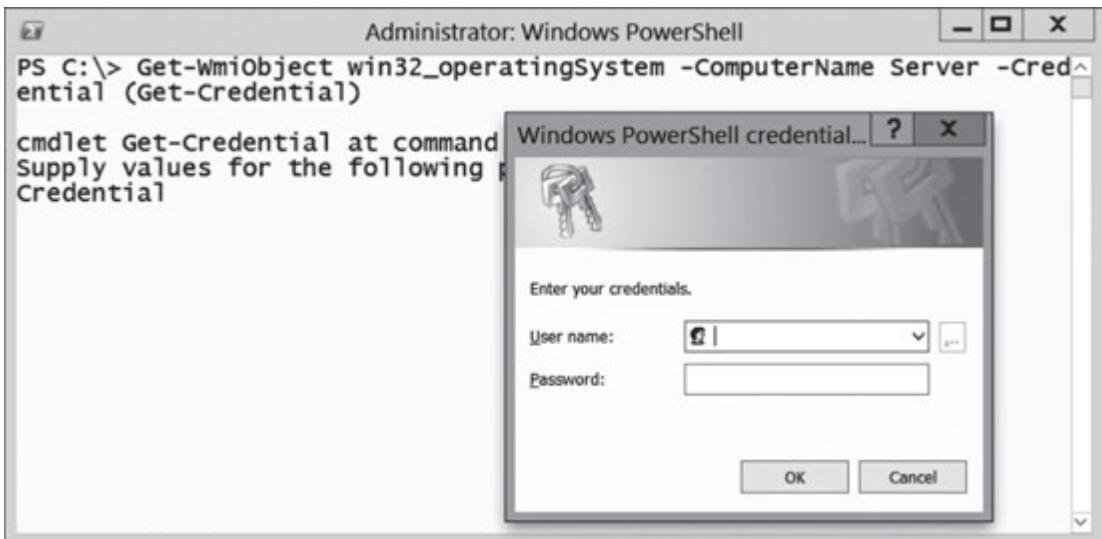


Figure 11-8

This example connects to computer Server and retrieves information on the **Win32\_Operatingsystem** WMI object. For alternate credentials, we call the **Get-Credential** cmdlet that presents what looks like a standard Windows authentication box. Except the **Get-Credential** cmdlet doesn't authenticate the user or verify the password—it simply stores this information securely in a **PSCredential** object.

When using the **CIM** cmdlets, you have to create a **CimSession** to take advantage of alternate credentials when using the **Get-CimInstance** cmdlet.

```
$CimSession = New-CimSession -ComputerName Server -Credential (Get-Credential)
Get-CimInstance -CimSession $CimSession -ClassName Win32_OperatingSystem
Remove-CimSession $CimSession
```

## VBScript alert

If you have a library of WMI scripts written in VBScript, don't delete them yet! With a little tweaking, you can take the essential WMI queries from your VBScripts and put them into PowerShell scripts. Use the **Where-Object** cmdlet for your WMI queries.

```
Select deviceID, drivetype, size, freespace from Win32_LogicalDisk where drivetype='3'
```

If you have a WMI query like the previous example, you could rewrite this in PowerShell as:

```
Get-WmiObject Win32_LogicalDisk | Select deviceid, drivetype, size, freespace | where { drivetype -eq 3 }
```

However, the better approach is to simply pass the original WQL query to **Get-WmiObject**, by using the **-Query** parameter.

```
Get-WmiObject -Query "Select deviceid, drivetype, size, freespace from Win32_LogicalDisk where drivetype=3"
```

Once you have the core query, you can tweak your PowerShell script and use the filtering, formatting, sorting, and exporting options that are available in PowerShell. If you don't have a script library, there are many, many WMI scripts available on the Internet that you can leverage and turn into PowerShell scripts.

The **Get-WmiObject** cmdlet does not allow you to specify multiple classes. So, if you want information from different Win32 classes, you'll have to call the cmdlet several times and store information in variables.

### **WMIReport.ps1**

```
#WMIReport.ps1
$OS=Get-WmiObject -Class Win32_OperatingSystem -Property Caption, CSDVersion

#select fixed drives only by specifying a drive type of 3
$Drives=Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType = 3"

Write-Host "Operating System:" $OS.Caption $OS.CSDVersion

Write-Host "Drive Summary:"
Write-Host "Drive`t"Size (MB)`t"FreeSpace (MB)"
foreach ($d in $Drives) {
    $free = "{0:N2}" -f ($d.FreeSpace/1MB -as [int])
    $size = "{0:N0}" -f ($d.size/1MB -as [int])
    Write-Host $d.deviceID `t $size `t $free
} #end foreach
```

This script reports system information from two WMI **Win32** classes. We first define a variable to hold operating system information, specifically the **Caption** and **CSDVersion**, which is the service pack. The second class, **Win32\_LogicalDisk**, is captured in **\$Drives**. Since we're only interested in fixed drives, we use the **-Filter** parameter to limit the query by drive type.

Once we have this information, we can display it any way we choose. Here's an example of what you might see when the script runs.

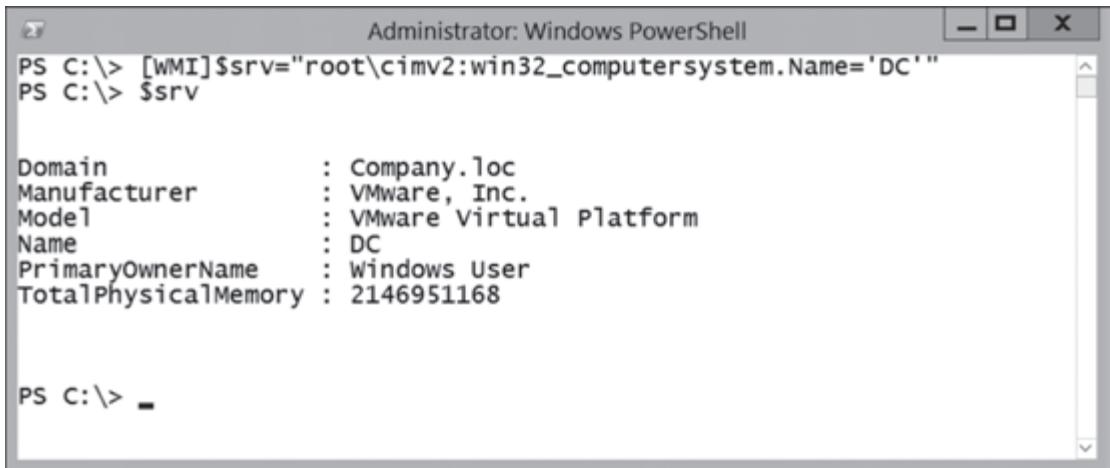
```
Operating System: Microsoft Windows Server 2012 R2 Datacenter
Drive Summary:
Drive      Size (MB)   FreeSpace (MB)
C:       61,088        50,335.00
```

Figure 11-9

## The [WMI] type

If you prefer a different approach to working with WMI and have some experience working with it, you may prefer to use PowerShell's **[WMI]** type.

```
[WMI]$srv = "root\cimv2:win32_computersystem.Name='DC'"
$srv
```



The screenshot shows an Administrator Windows PowerShell window. The command entered was `[WMI]$srv = "root\cimv2:win32_computersystem.Name='DC'"`. Then, `$srv` was executed, displaying the following properties:

Domain	:	Company.loc
Manufacturer	:	VMware, Inc.
Model	:	VMware Virtual Platform
Name	:	DC
PrimaryOwnerName	:	Windows User
TotalPhysicalMemory	:	2146951168

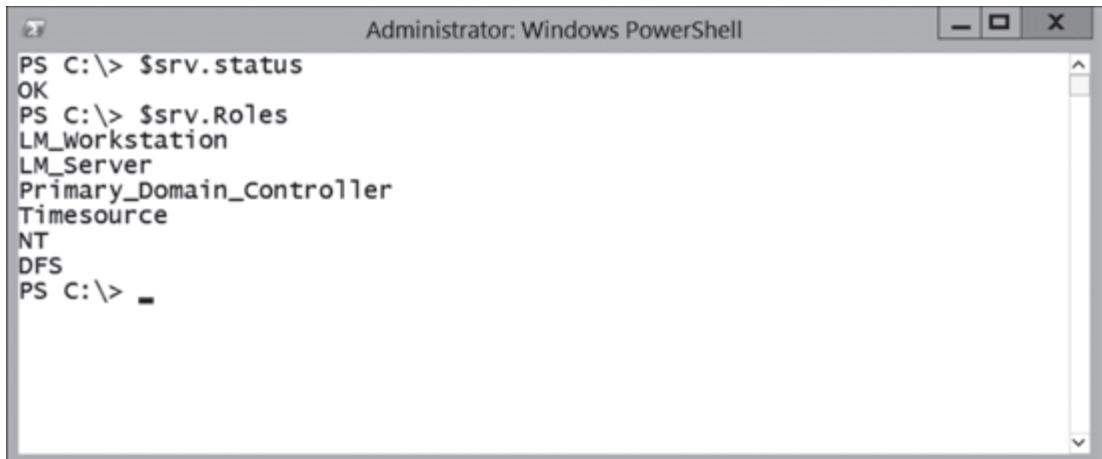
Figure 11-10

By creating a new object of type **[WMI]**, we can directly access the WMI instance. PowerShell returns a pre-defined subset of information when you call an object, as we did in Figure 11-10. However, you have access to all the WMI properties, which you can see by piping the object to the **Get-Member** cmdlet.

```
$srv | Get-Member
```

If we want additional information, all we have to do is check the object's properties.

```
$srv.Status
$srv.Roles
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The window shows the following command and its output:

```
PS C:\> $srv.status
OK
PS C:\> $srv.Roles
LM_Workstation
LM_Server
Primary_Domain_Controller
Timesource
NT
DFS
PS C:\> -
```

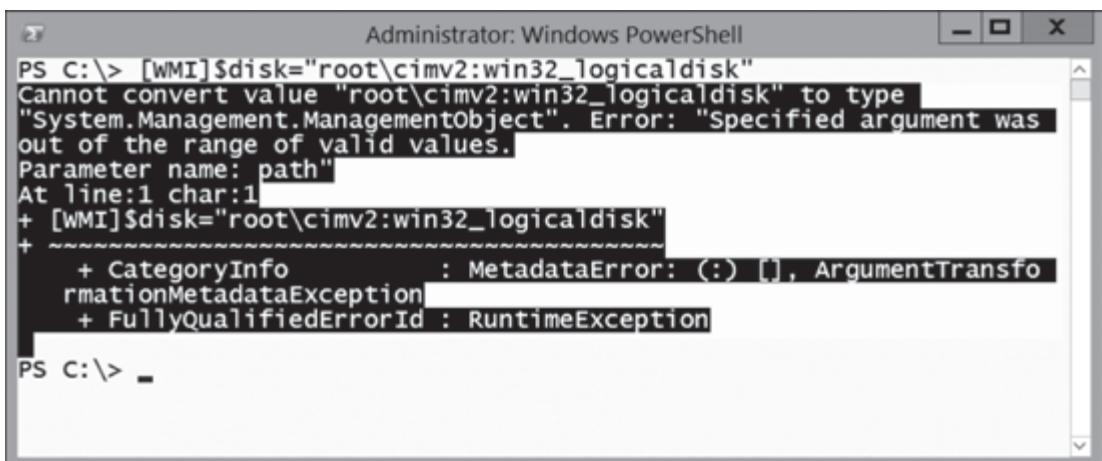
Figure 11-11

While you can't specify alternate credentials by using the [WMI] type, you can specify a remote system.

```
[WMI]$srv="\\Server\root\cimv2:win32_computersystem.Name='Server'"
```

To use the [WMI] type, you must create a reference to a specific instance of a WMI object. For example, you can't create a WMI object to return all instances of **Win32\_LogicalDisk**.

```
[WMI]$disk="root\cimv2:win32_logicaldisk"
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The window shows the following command and its error output:

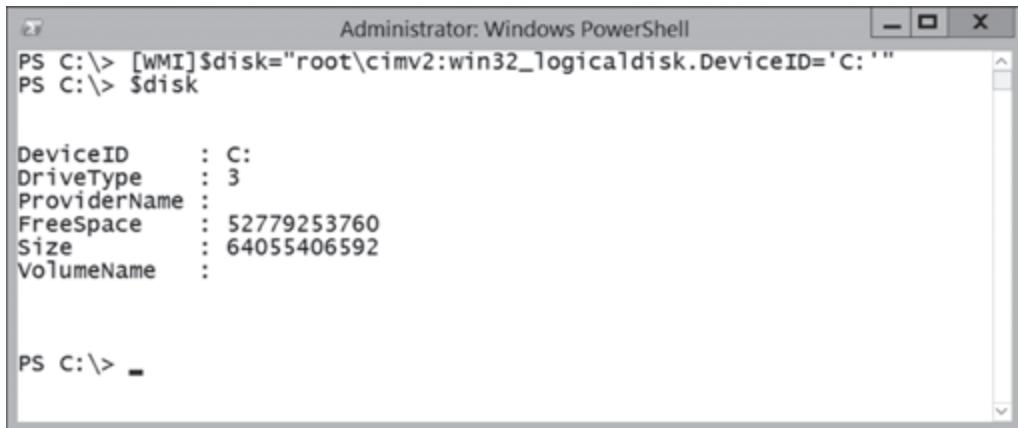
```
PS C:\> [WMI]$disk="root\cimv2:win32_logicaldisk"
Cannot convert value "root\cimv2:win32_logicaldisk" to type
"System.Management.ManagementObject". Error: "Specified argument was
out of the range of valid values.
Parameter name: path"
At line:1 char:1
+ [WMI]$disk="root\cimv2:win32_logicaldisk"
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransfo
rminationMetadataException
+ FullyQualifiedErrorId : RuntimeException
PS C:\> -
```

Figure 11-12

## Using WMI to manage systems

Instead, you must specify a single instance by using the WMI object's primary key and querying for a certain value.

```
[WMI]$disk="root\cimv2:win32_logicaldisk.DeviceID='C:'"
$disk
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is \$disk, which retrieves the properties of the C: drive. The output is as follows:

```
PS C:\> [WMI]$disk="root\cimv2:win32_logicaldisk.DeviceID='C:'"
PS C:\> $disk

DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 52779253760
Size          : 64055406592
VolumeName    :
```

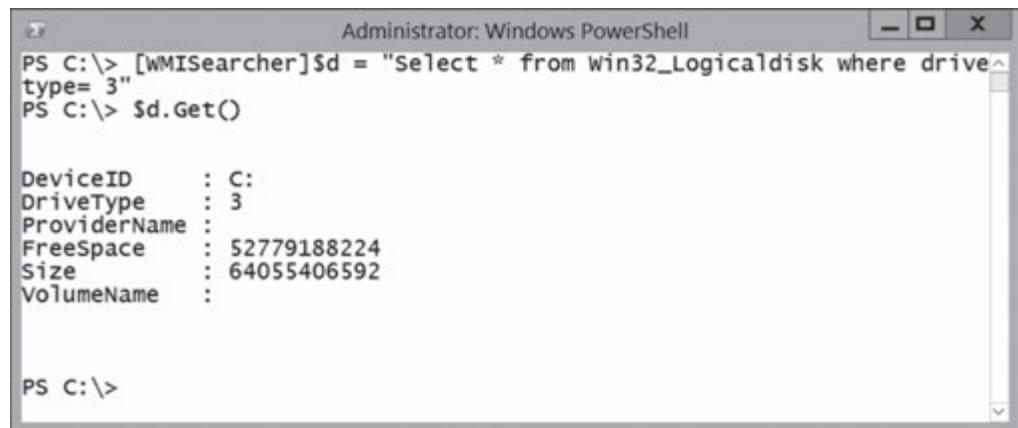
PS C:\> -

Figure 11-13

## The [WMISearcher] type

PowerShell also has a **[WMISearcher]** type. This allows you to submit a query to WMI and return a collection of objects.

```
[WMISearcher]$d = "Select * from Win32_Logicaldisk where drivetype= 3"
$d.Get()
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is \$d.Get(), which retrieves the properties of the C: drive using a WMI search. The output is as follows:

```
PS C:\> [WMISearcher]$d = "Select * from Win32_Logicaldisk where drivetype= 3"
PS C:\> $d.Get()

DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 52779188224
Size          : 64055406592
VolumeName    :
```

PS C:\>

Figure 11-14

The object, **\$d**, is a collection of all **Win32\_LogicalDisk** instances where drive type is equal to 3. To access the collection, we call the **Get()** method. This technique is very helpful when you want to find dynamic information that might include multiple instances of a given WMI class.

```
[WMISearcher]$s="Select * from win32_Service where StartMode='Disabled'"  
$s.Get() | Format-Table -AutoSize
```

ExitCode	Name	ProcessId	StartMode	State	Status
1077	AllUserInstallAgent	0	Disabled	Stopped	OK
1077	Browser	0	Disabled	Stopped	OK
1077	NetTcpPortSharing	0	Disabled	Stopped	OK
1077	NtFrs	0	Disabled	Stopped	OK
1077	RemoteAccess	0	Disabled	Stopped	OK
1077	SCardSrv	0	Disabled	Stopped	OK
1077	SharedAccess	0	Disabled	Stopped	OK
1077	SSDPSRV	0	Disabled	Stopped	OK
1077	upnphost	0	Disabled	Stopped	OK

Figure 11-15

If you want to query a remote system, you need to take an extra step to modify the management scope path.

```
$s.Scope.Path.Path=\\server\root\cimv2  
$s.Get() | Format-Table -AutoSize
```

So, when should you use the **Get-WmiObject/Get-CimInstance** cmdlets and when should you use the WMI type? If you know exactly the WMI object and instance you want to work with, and don't need to specify alternate credentials, then WMI type is the best and fastest approach. Otherwise, use the **Get-WmiObject** cmdlet.

## Invoking WMI methods

In PowerShell, there are two ways of calling a method of a WMI object. One is to use the **-ForEach-Object** cmdlet. We prefer this because it works with other cmdlets, not just WMI.

```
gwmi win32_process -Filter "name='notepad.exe'" | foreach { $_.terminate() }
```

PowerShell also includes the **Invoke-WmiMethod** cmdlet. This cmdlet shares many of the same parameters as **Get-WmiObject**. There are several ways you can use the cmdlet, depending on the method and objects you need to work with. Here's a simpler solution for our previous command.

```
gwmi win32_process -Filter "name='notepad.exe'" | Invoke-WmiMethod -Name terminate
```

Even if there are multiple instances of Notepad, they will all be terminated. You can also specify a particular WMI object, even on a remote computer. Let's change the browser service on a computer to have a **startmode** value of **manual**.

```
Invoke-WmiMethod -ComputerName Server -Credential $cred -Path "win32_service.name='browser'" -Name ChangeStartMode -ArgumentList "manual"
```

As you can see, we can include alternate credentials, as we've done by using a previously created **PSCredential** object. The path parameter is the WMI path to the object that you want to manage, in this case, the browser server. The name of the method we're invoking is **ChangeStartMode()**, which takes an argument indicating the new start mode. The WMI method returns a results object. A return value of 0 generally indicates success. Any other value means something went wrong. You'll need to turn to the WMI documentation for the class and method on MSDN to interpret these values.

## Practical examples

Let's look at a few practical examples of using WMI and PowerShell. If you are like most admins, you have a list or two of server names. Often, you want to obtain information from all of the servers in a list or perform some action against each of them, such as restarting a service. There are a few ways you can process multiple computers from a list.

```
foreach ($server in (Get-Content servers.txt)) {
    # do something else to each computer
    Get-WmiObject <WMIClass or query> -Computer $server
}
```

With this approach, you can run multiple commands on each server, passing the server name as the **-Computer** parameter for the **Get-WmiObject** cmdlet. Here's how this might look in a script.

```
foreach ($server in (Get-Content servers.txt)) {
    Write-Host $server.ToUpper() -Fore Black -Back Green
    Get-WmiObject Win32_Operatingsystem -Computer $server |
        Format-Table Caption, BuildNumber, ServicePackMajorVersion
    Get-WmiObject Win32_ComputerSystem -Computer $server |
        Format-Table Model, TotalPhysicalMemory
}
```

The code snippet takes each computer name from servers.txt by using the **Get-Content** cmdlet. Each time through the list, the computer name is assigned to the **\$server** variable and first displayed by using the **Write-Host** cmdlet.

```
Write-Host $server.ToUpper() -Fore Black -Back Green
```

Then we execute two different **Get-WmiObject** expressions and pipe the results of each to the **Format-Table** cmdlet. The end result is information about each operating system, including its name and service pack version, as well as information about the server model and the total amount of physical memory.

If you are only executing a single command for each server, here is a more efficient approach.

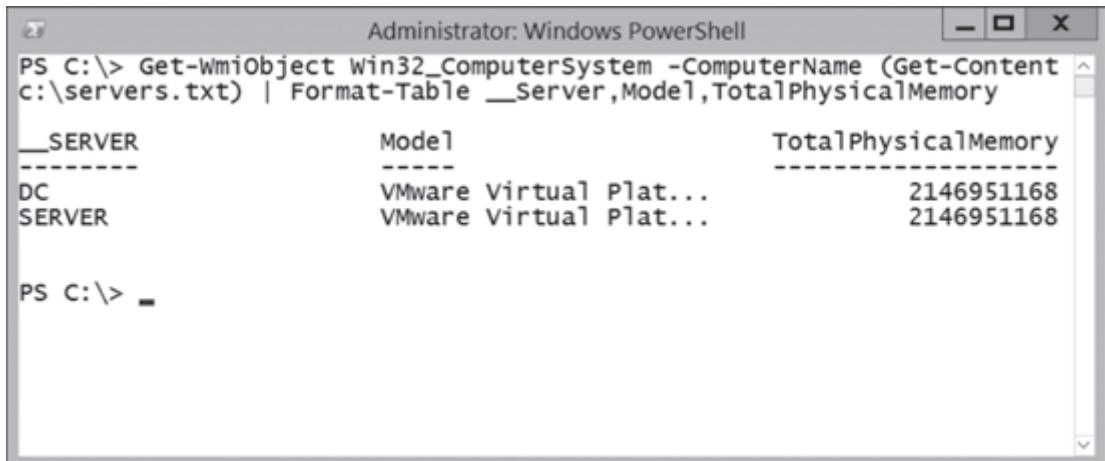
```
Get-WmiObject <WMI Query or Class> -Computer (Get-Content servers.txt)
```

You can use whatever syntax variation you want for the **Get-WmiObject** cmdlet. The trick here is that PowerShell will implicitly process the result of the **Get-Content** command as if you were using **ForEach**. Here's a variation on a previous example.

```
Get-WmiObject Win32_ComputerSystem -Computer (Get-Content servers.txt) | Format-Table Model, TotalPhysicalMemory
```

We will receive a table with hardware model and total physical memory for each computer in the list. However, if you run this yourself, you'll notice something is missing. How can you tell what information belongs to which computer? Easy. Have WMI tell you. Most WMI classes have a property, usually **CSName** or **MachineName**, which will return the name of the computer. Testing your WMI expression will let you know what you can use. If nothing else, you can always use the **\_\_SERVER** property—this is always populated. Here's our previous example, revised to include the server name.

```
Get-WmiObject Win32_ComputerSystem -ComputerName (Get-Content servers.txt) | Format-Table __Server, Model, TotalPhysicalMemory
```



```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject Win32_ComputerSystem -ComputerName (Get-Content c:\servers.txt) | Format-Table __Server,Model,TotalPhysicalMemory
__Server           Model          TotalPhysicalMemory
-----
DC                VMware Virtual Plat...    2146951168
SERVER            VMware Virtual Plat...    2146951168

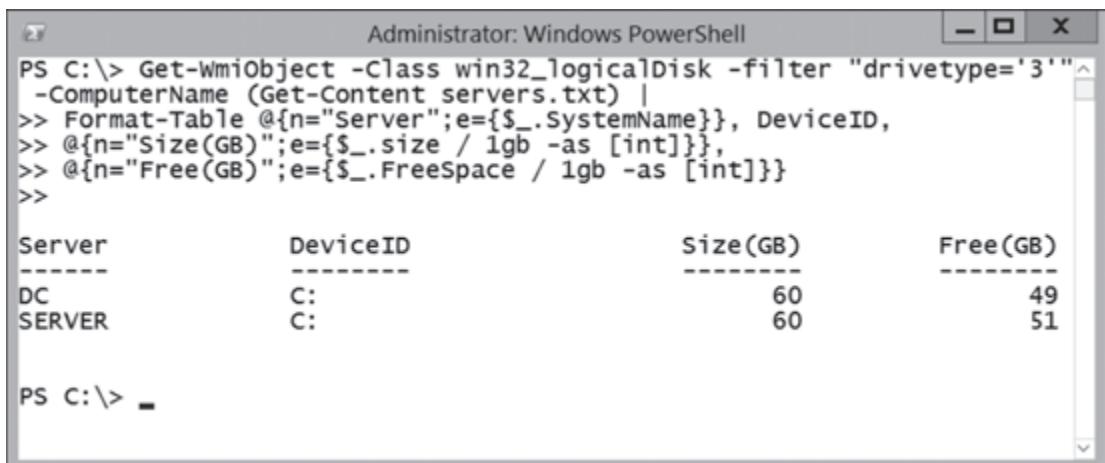
PS C:\> -
```

Figure 11-16

Now, you will generate a more meaningful report.

Here's one more slightly complex example, but something you are likely to need. This code sample could be used in a script to return drive utilization information.

```
Get-WmiObject -Class win32_logicalDisk -filter "drivetype='3'"
-ComputerName (Get-Content servers.txt) |
Format-Table @{n = "Server"; e = { $_.SystemName } }, DeviceID,
@{n = "Size(GB)"; e = { $_.size / 1gb -as [int] } },
@{n = "Free(GB)"; e = { $_.FreeSpace / 1gb -as [int] } }
```



```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject -Class win32_logicalDisk -filter "drivetype='3'"`n
-ComputerName (Get-Content servers.txt) |
>> Format-Table @{"n="Server";e={$_.SystemName}}, DeviceID,
>> @{$n="Size(GB)";e={$_.size / 1gb -as [int]}},
>> @{$n="Free(GB)";e={$_.Freespace / 1gb -as [int]}}
>>

Server        DeviceID      Size(GB)      Free(GB)
-----        -----      -----
DC            C:            60              49
SERVER        C:            60              51

PS C:\> -
```

Figure 11-17

The **Get-WmiObject** cmdlet is using the **-Query** parameter to return all logical disks of drive type 3, which indicates a fixed local disk. The example runs this query against each computer listed in servers.txt. You could also pass a stored set of alternate credentials if you needed, by using the **-Credential** parameter.

We then pipe the results of the query to the **Format-Table** cmdlet, which creates a few custom table headers. Since the query returned the **Size** and **FreeSpace** properties in bytes, we'll format them to gigabytes. The result is a table showing every server name, its fixed drives, and their size and free space in gigabytes.

## WMI events and PowerShell

A particularly useful feature of WMI is the ability to subscribe to Windows events and execute code in response to each event. For example, you could run a WMI script that would check every five seconds to verify that a specific service has not stopped. Or you might need to monitor a folder and be notified when a new file is created. WMI events are used to meet these needs. PowerShell includes cmdlets for working with WMI events.

## Register-WmiEvent

The **Register-WmiEvent** cmdlet creates a WMI subscriber for a defined WMI event. You can define the monitored event several ways. The easiest way is to specify a WMI trace class. Use the following expression to see the available classes.

```
gwmi -list | where { $_.name -match "trace" } | Select name
```

```
Administrator: Windows PowerShell
PS C:\> gwmi -list | where{$_ .name -match "trace"} | Select name
Name
-----
Win32_SystemTrace
Win32_ProcessTrace
Win32_ProcessStartTrace
Win32_ProcessStopTrace
Win32_ModuleTrace
Win32_ModuleLoadTrace
Win32_ThreadTrace
Win32_ThreadStartTrace
Win32_ThreadStopTrace
```

Figure 11-18

For example, suppose you want to watch for when a new process is started. We'll use the **Win32\_ProcessStartTrace** class.

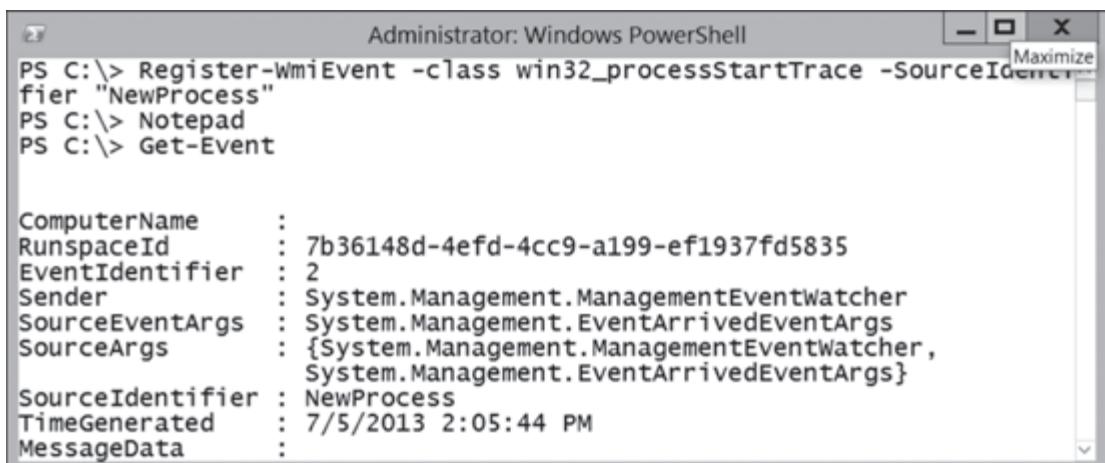
```
Register-WmiEvent -class 'Win32_ProcessStartTrace' -SourceIdentifier "NewProcess"
```

We also assign an identifier for this event, which will come in handy later. If you run this command, it appears that nothing happens. However, in the background, we created a WMI subscriber and any time a new process is launched and a corresponding event is fired, the subscriber captures it. Even though our examples are for the local machine, you can monitor events on remote computers as well, by using the **-ComputerName** parameter.

## Get-Event

Assuming you've started some processes or launched applications, you can run the **Get-Event** cmdlet to see the results.

```
Get-Event
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Register-WmiEvent -class win32\_processStartTrace -SourceIdentifier "NewProcess"" is run, followed by "Notepad" and "Get-Event". The output of "Get-Event" is displayed as a table:

	:
ComputerName	:
RunspaceId	: 7b36148d-4efd-4cc9-a199-ef1937fd5835
EventIdentifier	: 2
Sender	: System.Management.ManagementEventWatcher
SourceEventArgs	: System.Management.EventArrivedEventArgs
SourceArgs	: {System.Management.ManagementEventWatcher, System.Management.EventArrivedEventArgs}
SourceIdentifier	: NewProcess
TimeGenerated	: 7/5/2013 2:05:44 PM
MessageData	:

Figure 11-19

This will return all events. If you have multiple event subscribers, use the **-SourceIdentifier** parameter.

```
Get-Event -SourceIdentifier "NewProcess"
```

What is not easy to figure out is what process was started. The information is buried and hopefully will be easier to retrieve in future versions. Let's look at how to do it for a single event.

```
$p=(Get-Event)[0]
```

PowerShell embedded the process in the **NewEvent** property of the **SourceEventArgs** property.

```
$p.SourceEventArgs.NewEvent
```

```
Administrator: Windows PowerShell
PS C:\> $p=(Get-Event)[0]
PS C:\> $p.SourceEventArgs.NewEvent

__GENUS          : 2
__CLASS          : Win32_ProcessStartTrace
__SUPERCLASS     : Win32_ProcessTrace
__DYNASTY        : __SystemClass
__RELPATH         :
__PROPERTY_COUNT : 7
__DERIVATION      : {Win32_ProcessTrace, Win32_SystemTrace,
                     __ExtrinsicEvent, __Event...}
__SERVER          :
__NAMESPACE       :
__PATH            :
ParentProcessID  : 10772
ProcessID        : 10384
ProcessName       : notepad.exe
SECURITY_DESCRIPTOR:
SessionID         : 1
Sid               : {1, 5, 0, 0...}
TIME_CREATED      : 130175319440887536
PSComputerName    :
```

Figure 11-20

Armed with this information, it's simple to pull new process event information.

```
Get-Event -SourceIdentifier "newProcess" | Select TimeGenerated, @{n="Process"; E={ ($_.SourceEventArgs.NewEvent).processname}}
```

```

Administrator: Windows PowerShell
PS C:\> Get-Event -SourceIdentifier "NewProcess" | Select TimeGenerated, Process
TimeGenerated          Process
-----          -----
7/5/2013 2:05:44 PM      notepad.exe

PS C:\> _

```

Figure 11-21

## Remove-Event

The event queue will fill up pretty quickly. You can use the **Remove-Event** cmdlet to remove specific events or all events associated with a source identifier.

```
Remove-Event -SourceIdentifier "NewProcess"
```

## Unregister-Event

Even though you may have removed the event, this does not prevent additional events from queuing as they are fired. Doing so will require system resources, so if you no longer need to monitor the start process event, use **Unregister-Event**.

```
Unregister-Event -SourceIdentifier "NewProcess"
```

This snippet is only removing the **NewProcess** event subscription. If other subscriptions are defined, they will remain.

## Querying for specific events

If you need a more refined approach, you can also use a WMI query with the **Register-WmiEvent** cmdlet.

```
Register-WmiEvent -Query "Select * from win32_ProcessStartTrace where processname='notepad.exe'" -SourceIdentifier "New Notepad"
```

This creates a new event subscription that will only fire when Notepad is launched. The **Register-WmiEvent** cmdlet also has an **-Action** parameter, which will execute a defined script block when a targeted event fires, instead of sending it to the queue. Let's create an event subscription that watches for when Notepad.exe ends.

```
Register-WmiEvent -Query "Select * from win32_ProcessStopTrace where processname='notepad.exe'" -sourceIdentifier "Close Notepad" -Action {("{0} Notepad has terminated" -f (Get-Date)) | Add-Content c:\log.txt}
```

This expression is watching for events where Notepad.exe is terminated. When the event fires, the action script block instructs PowerShell to write information to a log file. This will continue until you unregister the “Close Notepad” event.

## Watching services

Let's look at another scenario and another technique. You may want to watch for when a mission-critical service stops. There are no trace classes for service objects, but there are other classes we can use. WMI has several system classes that we can leverage.

- `__InstanceCreationEvent`
- `__InstanceModificationEvent`
- `__InstanceDeletionEvent`

The names should give you a good idea of what to expect. These are very broad events, so we need to build a more refined query. When one of these events fires, a new object, the **TargetInstance**, is also created as a returned property. In our situation, the target instance will be a **Win32\_service** object. Let's further stipulate that we want to watch for changes to the Secondary Logon service. Here is what our query might look like.

```
Select * from __InstanceModificationEvent where TargetInstance ISA 'win32_service' and name='seclogon'
```

In practice though, we probably don't need to be constantly checking for when an event fires. Instead, we will use a polling interval to tell WMI to check every x number of seconds. Here's the revised query.

```
$query = "Select * from __InstanceModificationEvent within 10 where TargetInstance ISA 'win32_service' and Targetinstance.name='seclogon'"
```

Now let's use it with the **Register-WmiEvent** cmdlet.

```
Register-WmiEvent -Query $query -SourceIdentifier "Service Change" -ComputerName "Server"
```

If we modify the service in some way, the event will fire and we can retrieve it with the **Get-Event** cmdlet.

```
Get-Event -SourceIdentifier "Service Change"
```

We already know what service was changed. But suppose we didn't? We also can't tell from this what change was made. Notice the **TargetInstance** property? There is also a **PreviousInstance** property that holds the object's previous state. Comparing the two objects isn't too difficult, although it does require a little finesse.

```
$event = (Get-Event -SourceIdentifier "Service Change").SourceEventArgs.newEvent
$previous = $event.PreviousInstance
$target = $event.TargetInstance
foreach ($property in ($previous.psbase.properties)) {
    if ($previous.($property.name) -ne $target.($property.name))
    {
        $obj=New-Object PSObject
        $obj | Add-Member NoteProperty "Property" $property.name
        $obj | Add-Member NoteProperty "Previous" $previous.($property.name)
        $obj | Add-Member NoteProperty "Target" $target.($property.name)
        write $obj
    }
}
```

This could probably be written more concisely, but we didn't want to sacrifice the clarity. We first create a variable for the event object. Remember, this has previous and target instance properties, which we also save to variables. The **ForEach** construct enumerates the properties of the previous instance, which should be identical in name, but not value, to the target instance. Within the enumeration, we compare the property on both objects. If they are different, we create a custom object and add some properties so that we can see what properties changed between the two objects.

## Watching files and folders

We'll wrap up our discussion of WMI events with another common request we've seen: notification when a folder changes. Usually the requirement is to watch for new files and take action when they are detected. Here's how we might solve this problem.

We will use a WMI query for the **\_InstanceCreationEvent** class, where the target instance is a **CIM\_DATAFILE**. This class can be tricky to work with and in our experience, the more specific you can make your query, the better.

```
$computername="Server"
$drive = "c:"
#folder path must escape the \ with another \
$folder = "\\test\\files\\"
$poll = 10
$query = "Select * from __InstanceCreationEvent Within $poll where TargetInstance ISA 'CIM_datafile' AND TargetInstance.drive='$drive' AND TargetInstance.Path='$folder'"
```

```
Register-WmiEvent -Query $query -SourceIdentifier "New File" -MessageData "File Created"  
-ComputerName $computer
```

When the script creates a new file in C:\test\files, it fires an event that the event subscriber receives, which you can view with **Get-Event**.

As before, we need to dig a little to get information about the target instance.

```
Get-Event -SourceIdentifier "New File" | foreach {  
    $_.sourceEventArgs.NewEvent.TargetInstance  
}
```

Use the standard PowerShell cmdlets to filter, select, or sort the output.

The **Register-WmiEvent** cmdlet has a handy parameter for this specific situation: **-Action**. You might want to add code to copy the new file or perhaps send an email notification. The possibilities are limited only by your creativity.

## Read more about it

Using WMI, including the new CIM cmdlets, in practice is a daunting task. There are so many namespaces and classes. Where do you start? It helps to start exploring WMI, with a tool such as SAPIEN'S WMI Explorer, and there are several books dedicated to using WMI that point you directly to the class you need.

We recommend you use the WMI Explorer and your favorite search engine to find the class you need. As an example, you can open your browser and type “List services with WMI”, and you will find the class that will help with that. The search function in the WMI Explorer is better than an Internet search because it focuses only on WMI.

Whatever your approach, WMI requires some research. Don’t let that slow you down in finding, or changing, the information you need!

## In Depth 12

# Object serialization

### Object serialization

Occasionally, there is a need for objects to be represented in a more easily portable format, such as XML. Serialization is the process of taking an object and converting it into an XML representation. The reverse, deserialization, converts the XML back into an object—although the object is often less functional than it was prior to serialization, often including only property values and omitting methods since it is no longer connected to the real-world software that originally generated it. In other words, if you serialize a Windows service into an XML file, you can carry that to another computer and deserialize it back into an object. Yet that object won't be able to start and stop the original service—it'll simply be a way of examining the service's properties as they were at the time it was serialized. Serialized objects, then, are essentially a snapshot of an object at a specific point in time.

Windows PowerShell primarily uses the **Export-Clixml** cmdlet to serialize objects and save the resulting XML in a text file.

```
Get-WmiObject -Class Win32_OperatingSystem |  
    Export-Clixml C:\test\win32os.xml
```

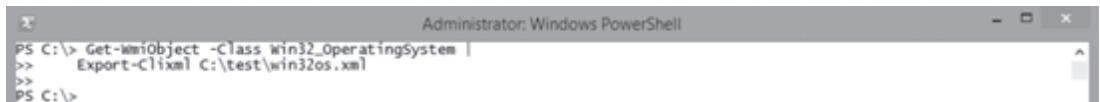
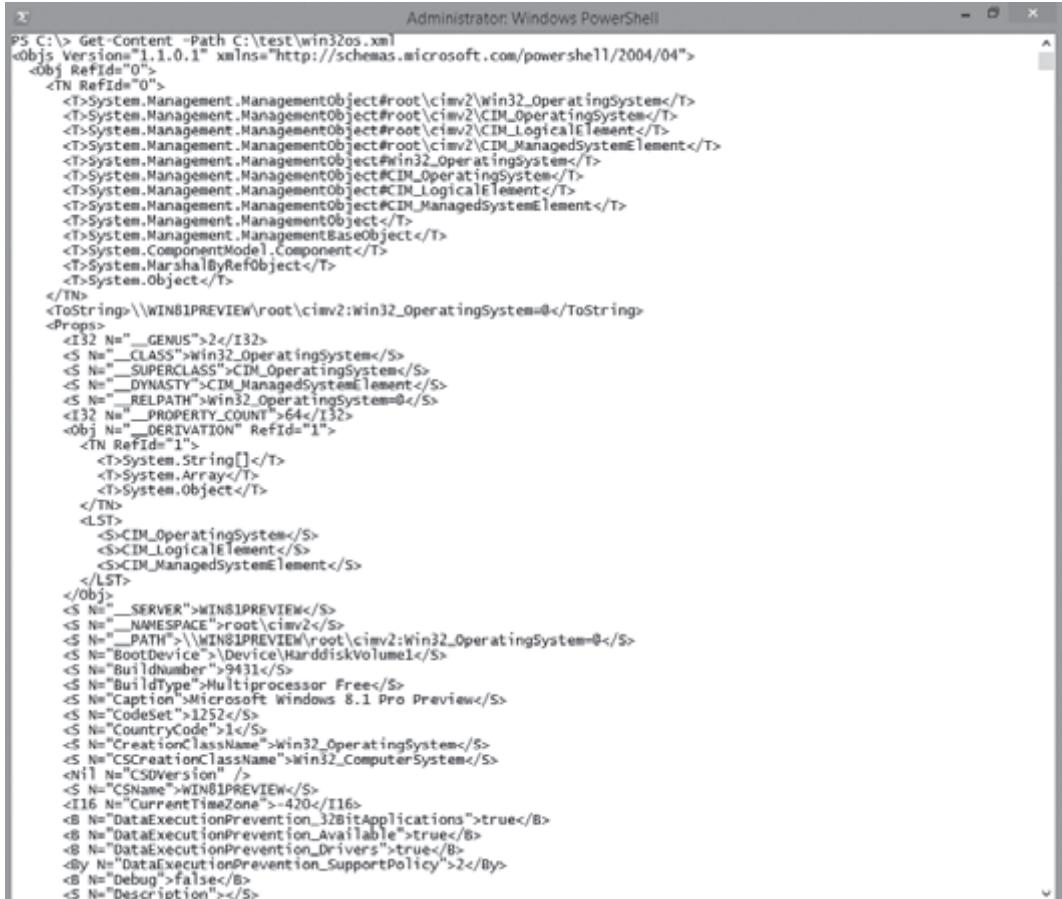
A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The window shows the command "Get-WmiObject -Class Win32\_OperatingSystem | Export-Clixml C:\test\win32os.xml" being typed in. The command has been partially entered, with the final line starting with "PS C:\>". The background of the window is white, and the text is black.

Figure 12-1

## Windows PowerShell: TFM

The previous command results in the following XML representation (which we've truncated to save space).

```
Get-Content -Path C:\test\win32os.xml
```

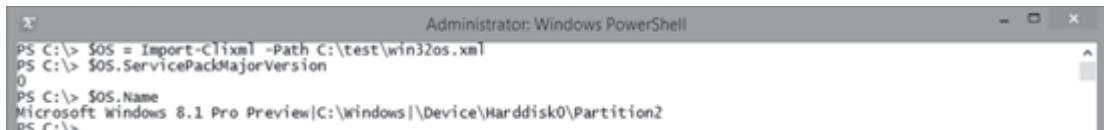


```
P5 C:\> Get-Content -Path C:\test\win32os.xml
Administrator: Windows PowerShell
<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">
<Obj RefId="0">
<TN RefId="0">
<T>System.Management.ManagementObject#root\cimv2!Win32_OperatingSystem</T>
<T>System.Management.ManagementObject#root\cimv2!CIM_OperatingSystem</T>
<T>System.Management.ManagementObject#root\cimv2!CIM_LogicalElement</T>
<T>System.Management.ManagementObject#root\cimv2!CIM_ManagedSystemElement</T>
<T>System.Management.ManagementObject#Win32_OperatingSystem</T>
<T>System.Management.ManagementObject#CIM_OperatingSystem</T>
<T>System.Management.ManagementObject#CIM_LogicalElement</T>
<T>System.Management.ManagementObject#CIM_ManagedSystemElement</T>
<T>System.Management.ManagementObject</T>
<T>System.Management.ManagementBaseObject</T>
<T>System.ComponentModel.Component</T>
<T>System.MarshalByRefObject</T>
<T>System.Object</T>
</TN>
<ToString>\WIN81PREVIEW\root\cimv2:Win32_OperatingSystem=@</ToString>
<Props>
<I32 Nm="__GENUS">2</I32>
<S Nm="__CLASS">Win32_OperatingSystem</S>
<S Nm="__SUPERCLASS">CIM_OperatingSystem</S>
<S Nm="__DYNASTY">CIM_ManagedSystemElement</S>
<S Nm="__RELPATH">Win32_OperatingSystem=@</S>
<I32 Nm="__PROPERTY_COUNT">64</I32>
<Obj Nm="__DERIVATION" RefId="1">
<TN RefId="1">
<T>System.String[]</T>
<T>System.Array</T>
<T>System.Object</T>
</TN>
<LST>
<S>CIM_OperatingSystem</S>
<S>CIM_LogicalElement</S>
<S>CIM_ManagedSystemElement</S>
</LST>
</Obj>
<S Nm="__SERVER">WIN81PREVIEW</S>
<S Nm="__NAMESPACE">root\cimv2</S>
<S Nm="__PATH">\WIN81PREVIEW\root\cimv2:Win32_OperatingSystem=@</S>
<S Nm="BootDevice">\Device\HarddiskVolume1</S>
<S Nm="BuildNumber">9431</S>
<S Nm="BuildType">Multiprocessor Free</S>
<S Nm="Caption">Microsoft Windows 8.1 Pro Preview</S>
<S Nm="CodeSet">1252</S>
<S Nm="CountryCode">1</S>
<S Nm="CreationClassName">Win32_OperatingSystem</S>
<S Nm="CSCreationClassName">Win32_ComputerSystem</S>
<Nil Nm="CSVersion" />
<S Nm="CSName">WIN81PREVIEW</S>
<I16 Nm="CurrentTimeZone">-420</I16>
<B Nm="DataExecutionPrevention_32BitApplications">true</B>
<B Nm="DataExecutionPrevention_Available">true</B>
<B Nm="DataExecutionPrevention_Drivers">true</B>
<By Nm="DataExecutionPrevention_SupportPolicy">2</By>
<S Nm="Debug">false</S>
<S Nm="Description"></S>
```

Figure 12-2

The **Import-Clixml** cmdlet does the opposite, returning the XML to a static object inside the shell.

```
$OS = Import-Clixml -Path C:\test\win32os.xml
$OS.ServicePackMajorVersion
$OS.Name
```



```
Administrator: Windows PowerShell
PS C:\> $OS = Import-Clixml -Path C:\test\win32os.xml
PS C:\> $OS.ServicePackMajorVersion
0
PS C:\> $OS.Name
Microsoft Windows 8.1 Pro Preview|C:\Windows|\Device\Harddisk0\Partition2
PS C:\> _
```

Figure 12-3

## Why export objects to XML?

Exporting, or serializing, objects to XML allows them to be persisted, or saved, as a static snapshot. One practical reason for doing this is to share those objects with other PowerShell users. For example, you might want to export your command-line history to an XML file so that you can share it with another user—who could then import it to re-create your command-line history.

Another less obvious reason might be to retrieve a snapshot of objects when you're not physically around. For example, suppose you have a long-running process that starts on one of your servers at 01:00 daily. You know it should finish by 05:00. You could write a very short PowerShell script to handle this task.

```
Get-Process | Export-Clixml -Path C:\test\1am.xml
```



```
Administrator: Windows PowerShell
PS C:\> Get-Process | Export-Clixml -Path C:\test\1am.xml
PS C:\> _
```

Figure 12-4

And you could schedule it to run at 01:15, when the long-running process should be running. Later, at 05:30, you could run a second script.

```
Get-Process | Export-Clixml -Path C:\test\5am.xml
```



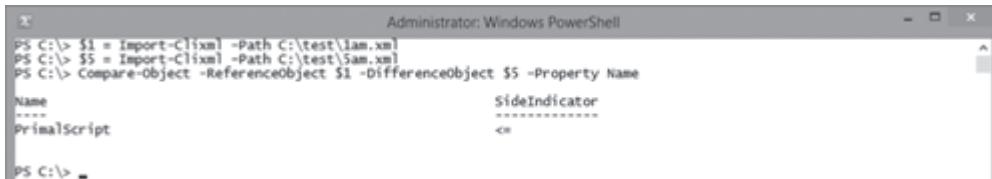
```
Administrator: Windows PowerShell
PS C:\> Get-Process | Export-Clixml -Path C:\test\5am.xml
PS C:\> _
```

Figure 12-5

When you arrive for work, you could grab both of these files and re-import them, effectively reconstructing the objects as they were when the XML file was created. This would let you examine the objects from that point in time, even though you weren't physically present then. For example, you can compare the two sets of objects.

## Windows PowerShell: TFM

```
$1 = Import-Clixml -Path C:\test\1am.xml
$5 = Import-Clixml -Path C:\test\5am.xml
Compare-Object -ReferenceObject $1 -DifferenceObject $5 -Property Name
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$1 = Import-Clixml -Path C:\test\1am.xml is entered. The command PS C:\> \$5 = Import-Clixml -Path C:\test\5am.xml is entered. The command PS C:\> Compare-Object -ReferenceObject \$1 -DifferenceObject \$5 -Property Name is entered. The output shows a table with columns "Name" and "SideIndicator". The "Name" column lists "PrimalScript". The "SideIndicator" column shows "<=>".

Figure 12-6

Using this example, the **\$1** variable contains all of the objects that were running at 01:00. You can pipe **\$1** to any other cmdlet capable of working with objects, allowing you to sort, group, filter, or format the process objects in any way. This ability to easily persist objects—a result of PowerShell's serialization capabilities—has countless uses.

Serialization is a way of saving objects into a simplified, easily transportable format. For example, you might export some objects from one computer, move the XML file to another computer, and then import the objects from XML to work with them again. Saving objects is another good use of serialization. For example, by piping the **Get-History** cmdlet to the **Export-CliXML** cmdlet, you can save your command history in an XML file. You can then use the **Import-CliXML** cmdlet to import that file, pipe it to the **Add-History** cmdlet, and then reload your command history. This is useful when giving demonstrations, or when conducting various repetitive tasks.

```
Get-History | Export-Clixml -Path C:\test\history.xml
Clear-History
Import-Clixml -Path C:\test\history.xml | Add-History
Get-History
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Get-History | Export-Clixml -Path C:\test\history.xml is entered. The command PS C:\> Clear-History is entered. The command PS C:\> Import-Clixml -Path C:\test\history.xml | Add-History is entered. The command PS C:\> Get-History is entered. The output shows a large list of command history entries, including various commands like Get-WmiObject, Get-Content, Invoke-Command, and ping, along with their parameters and results.

Figure 12-7

In Figure 12-7, we cleared our history with the **Clear-History** cmdlet between the commands to simulate closing PowerShell or moving the XML file to a different computer. We also ran the **Get-History** cmdlet to show that our history was indeed imported.

## Jobs

When a cmdlet is run as a job, the results are deserialized. This occurs even when the target computer of the job is the local computer, as shown in Figure 12-8.

```
Start-Job -Name BitsService -ScriptBlock { Get-Service -Name BITS }
Receive-Job -Name BitsService | Get-Member
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Start-Job -Name BitsService -ScriptBlock { Get-Service -Name BITS } is run, followed by PS C:\> Receive-Job -Name BitsService | Get-Member. The output shows a table of job properties and their types, and a detailed list of service controller properties.

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
4	BitsService	BackgroundJob	Running	True	localhost	Get-Service -Name BITS

```
PS C:\> Receive-Job -Name BitsService | Get-Member

 TypeName: Deserialized.System.ServiceProcess.ServiceController

Name          MemberType  Definition
----          --          --
ToString      Method     string ToString(), string ToString(string format, System.IFormatProvider formatProv...
Name          NoteProperty System.String Name=BITS
PSCoputerName NoteProperty System.String PSCoputerName=localhost
PSShowComputerName NoteProperty System.Boolean PSShowComputerName=False
RequiredServices NoteProperty Deserialized.System.ServiceProcess.ServiceController[] RequiredServices=RpcSs Event...
RunspaceId    NoteProperty System.Guid RunspaceId=3909b3a1-3142-4429-a1d8-d319d4d7a2e3
CanPauseAndContinue Property   System.Boolean {get;set;}
CanShutdown   Property   System.Boolean {get;set;}
CanStop      Property   System.Boolean {get;set;}
Container     Property   {get;set;}
DependentServices Property  Deserialized.System.ServiceProcess.ServiceController[] {get;set;}
DisplayName   Property   System.String {get;set;}
MachineName   Property   System.String {get;set;}
ServiceHandle Property   System.String {get;set;}
ServiceName   Property   System.String {get;set;}
ServicesDependedOn Property Deserialized.System.ServiceProcess.ServiceController[] {get;set;}
ServiceType   Property   System.String {get;set;}
Site         Property   {get;set;}
Status       Property   System.String {get;set;}
```

Figure 12-8

## Remote management

Serialization plays an important role in remote management. PowerShell allows you to connect to remote copies of the shell via what's called PowerShell remoting and to have commands execute locally on the remote computer. Those remote shells serialize the results of your commands, transmit the results—as a stream of XML text via an HTTP-like connection—back to you, where your shell reconstructs the objects so that you can work with them. The deserialized objects that are returned from a command that was run via the PowerShell remoting **Invoke-Command** cmdlet are shown in Figure 12-9.

```
Invoke-Command -ComputerName dc02 {
    Get-Service -Name BITS
} |
Get-Member
```

```
Administrator: Windows PowerShell
PS C:\> Invoke-Command -ComputerName dc02 {
>>     Get-Service -Name BITS
>> } |
>>     Get-Member
>>

TypeName: Deserialized.System.ServiceProcess.ServiceController
Name      MemberType  Definition
----      -----  -----
GetType   Method     type GetType()
ToString  Method     string ToString(string format, System.IFormatProvider formatProv...
Name     NoteProperty System.String Name=BITS
PSCoputerName NoteProperty System.String PSCoputerName=dc02
PSShowComputerName NoteProperty System.Boolean PSShowComputerName=True
RequiredServices NoteProperty Deserialized.System.ServiceProcess.ServiceController[] RequiredServices=RpcSs Event...
RunspaceId NoteProperty System.Guid RunspaceId=5d9ae45F-23b1-464c-b990-f3d8ca29c719
CanPauseAndContinue Property  System.Boolean {get;set;}
CanShutdown  Property  System.Boolean {get;set;}
CanStop     Property  System.Boolean {get;set;}
Container   Property  {get;set;}
DependentServices Property Deserialized.System.ServiceProcess.ServiceController[] {get;set;}
DisplayName  Property  System.String {get;set;}
MachineName  Property  System.String {get;set;}
ServiceHandle Property  System.String {get;set;}
ServiceName  Property  System.String {get;set;}
ServicesDependedOn Property Deserialized.System.ServiceProcess.ServiceController[] {get;set;}
ServiceType   Property  System.String {get;set;}
Site        Property  {get;set;}
Status      Property  System.String {get;set;}
```

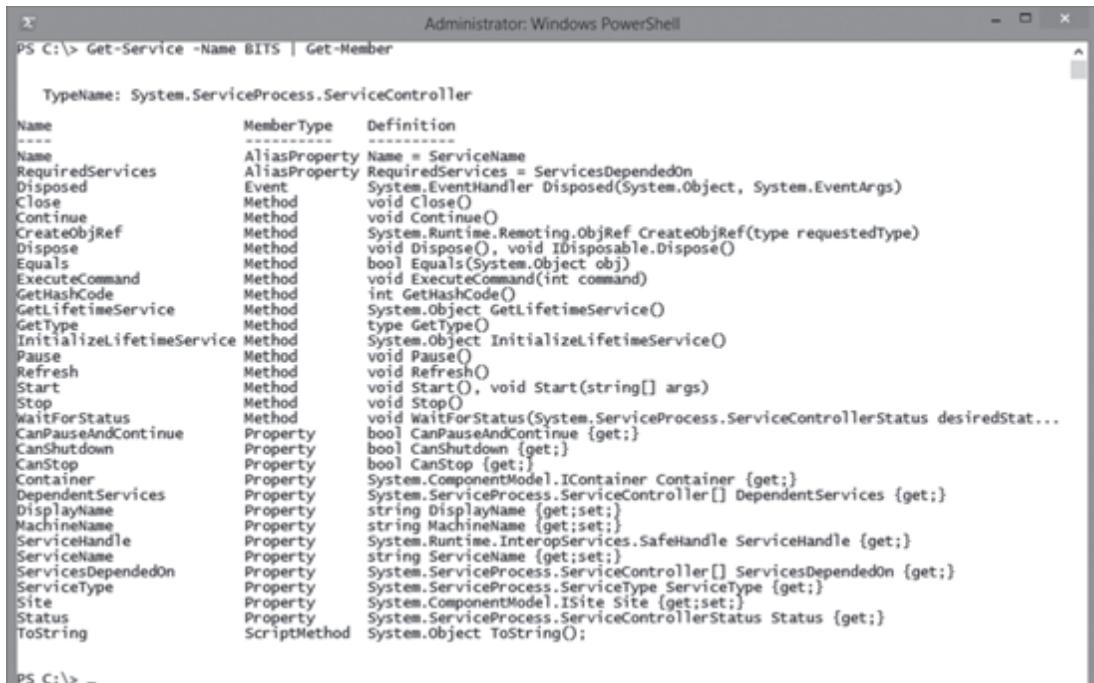
Figure 12-9

Compare the methods that are available from the results in Figure 12-9, which were run against a remote computer via PowerShell remoting, to the methods that are available in the command that's run in the next example, which was run against the local computer. In Figure 12-10, note the difference in the **TypeName**, which tells us the type of object returned.

Comparing these results makes it easy to see that the objects returned by a command that was run via PowerShell remoting are inert, since the methods that allow us to take action on the object are clearly missing.

```
Get-Service -Name BITS | Get-Member
```

## Object serialization



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Get-Service -Name BITS | Get-Member". The output displays the type information for the System.ServiceProcess.ServiceController class, listing various properties and methods along with their definitions.

```

TypeName: System.ServiceProcess.ServiceController

Name          MemberType  Definition
----          --          --
Name          AliasProperty  Name = ServiceName
RequiredServices AliasProperty  RequiredServices = ServicesDependedOn
Disposed       Event        System.EventHandler Disposed(System.Object, System.EventArgs)
Close         Method      void Close()
Continue     Method      void Continue()
CreateObjRef  Method      System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose       Method      void Dispose(), void IDisposable.Dispose()
Equals        Method      bool Equals(System.Object obj)
ExecuteCommand Method      void ExecuteCommand(int command)
GetHashCode   Method      int GetHashCode()
GetLifetimeService Method      System.Object GetLifetimeService()
GetType       Method      type GetType()
InitializeLifetimeService Method      System.Object InitializeLifetimeService()
Pause         Method      void Pause()
Refresh       Method      void Refresh()
Start         Method      void Start(), void Start(string[] args)
Stop          Method      void Stop()
WaitForStatus Method      void WaitForStatus(System.ServiceProcess.ServiceControllerStatus desiredStat...
CanPauseAndContinue Property    bool CanPauseAndContinue {get;}
CanShutdown   Property    bool CanShutdown {get;}
CanStop       Property    bool CanStop {get;}
Container     Property    System.ComponentModel.IContainer Container {get;}
DependentServices Property    System.ServiceProcess.ServiceController[] DependentServices {get;}
DisplayName   Property    string DisplayName {get;set;}
MachineName   Property    string MachineName {get;set;}
ServiceHandle Property    System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName   Property    string ServiceName {get;set;}
ServicesDependedOn Property    System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType   Property    System.ServiceProcess.ServiceType ServiceType {get;}
Site          Property    System.ComponentModel.ISite Site {get;set;}
Status        Property    System.ServiceProcess.ServiceControllerStatus Status {get;}
ToString      ScriptMethod System.Object ToString();

```

Figure 12-10

If you need to access the object's methods, that will need to be done in the remoting session. The action is performed on the remote computer where the command is actually being executed, so technically the method is being accessed locally where the remote command is being executed.

Figure 12-11 shows an example of the **Stop** method being executed for the **bits** service in a PowerShell remoting session.

```
Invoke-Command -ComputerName dc02 {
(Get-Service -Name BITS).stop()}
```



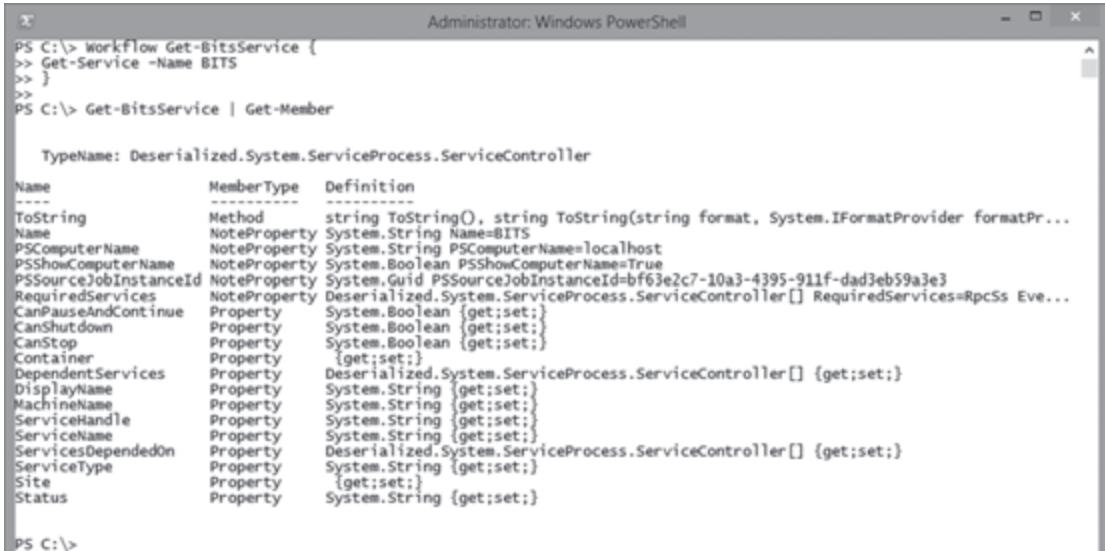
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is "PS C:\> Invoke-Command -ComputerName dc02 {>> (Get-Service -Name BITS).stop()}>>". This command uses the Invoke-Command cmdlet to run the Stop() method on the BITS service on the remote computer dc02.

Figure 12-11

## Workflow

By default, an activity in a workflow returns deserialized objects. Here is a simple workflow to demonstrate this.

```
Workflow Get-BitsService {
    Get-Service -Name BITS
}
Get-BitsService | Get-Member
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Workflow Get-BitsService {>> Get-Service -Name BITS>>}>> PS C:\> Get-BitsService | Get-Member is run. The output shows the properties of the BITS service controller object:

Name	MemberType	Definition
ToString	Method	string ToString(), string ToString(string format, System.IFormatProvider formatPr...)
Name	NoteProperty	System.String Name=BITS
PSCoputerName	NoteProperty	System.String PSCoputerName=localhost
PSShowComputerName	NoteProperty	System.Boolean PSShowComputerName=True
PSSourceJobInstanceId	NoteProperty	System.Guid PSSourceJobInstanceId=fb63e2c7-10a3-4395-911f-dad3eb59a3e3
RequiredServices	NoteProperty	Deserialized.System.ServiceProcess.ServiceController[] RequiredServices=RpcSs Eve...
CanPauseAndContinue	Property	System.Boolean {get;set;}
CanShutdown	Property	System.Boolean {get;set;}
CanStop	Property	System.Boolean {get;set;}
Container	Property	{get;set;}
DependentServices	Property	Deserialized.System.ServiceProcess.ServiceController[] {get;set;}
DisplayName	Property	System.String {get;set;}
MachineName	Property	System.String {get;set;}
ServiceHandle	Property	System.String {get;set;}
ServiceName	Property	System.String {get;set;}
ServicesDependedOn	Property	Deserialized.System.ServiceProcess.ServiceController[] {get;set;}
ServiceType	Property	System.String {get;set;}
Site	Property	{get;set;}
Status	Property	System.String {get;set;}

Figure 12-12

**Workflow** includes a **-PSDisableSerialization** parameter, which causes an activity to return live objects. So when this parameter is specified and is given a value of **True**, the objects have methods.

```
Workflow Get-BitsService {
    Get-Service -Name BITS -PSDisableSerialization $true
}
Get-BitsService | Get-Member
```

## Object serialization

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Workflow Get-BitsService { >> Get-Service -Name BITS -PSDisableSerialization \$true >> } >> | Get-Member is run. The output shows the members of the System.ServiceProcess.ServiceController type, including methods like Close(), Continue(), Dispose(), Equals(), ExecuteCommand(), GetHashCode(), GetLifetimeService(), GetType(), InitializeLifetimeService(), Pause(), Refresh(), Start(), Stop(), and WaitForStatus(). Note that the PSDisableSerialization parameter is set to \$true, which prevents live methods from being returned by activities in a workflow.

Name	MemberType	Definition
====	=====	=====
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDependedOn
Disposed	Event	System.EventHandler Disposed(System.Object, System.EventArgs)
Close	Method	void Close()
Continue	Method	void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose	Method	void Dispose(), void IDisposable.Dispose()
Equals	Method	bool Equals(System.Object obj)
ExecuteCommand	Method	void ExecuteCommand(int command)
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifetimeService()
Pause	Method	void Pause()
Refresh	Method	void Refresh()
Start	Method	void Start(), void Start(string[] args)
Stop	Method	void Stop()
WaitForStatus	Method	void WaitForStatus(System.ServiceProcess.ServiceControllerStatus desiredStat...
PSComputerName	NoteProperty	System.String PSComputerName=localhost
PSShowComputerName	NoteProperty	System.Boolean PSShowComputerName=True
PSSourceJobInstanceId	NoteProperty	System.Guid PSSourceJobInstanceId=390b3882-05fd-475e-b227-269aaeb431a3
CanPauseAndContinue	Property	bool CanPauseAndContinue {get;}
Canshutdown	Property	bool CanShutdown {get;}
CanStop	Property	bool CanStop {get;}
Container	Property	System.ComponentModel.IContainer Container {get;}
DependentServices	Property	System.ServiceProcess.ServiceController[] DependentServices {get;}
DisplayName	Property	string DisplayName {get;set;}
MachineName	Property	string MachineName {get;set;}
ServiceHandle	Property	System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName	Property	string ServiceName {get;set;}
ServicesDependedOn	Property	System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType	Property	System.ServiceProcess.ServiceType ServiceType {get;}
Site	Property	System.ComponentModel.ISite Site {get;set;}
Status	Property	System.ServiceProcess.ServiceControllerStatus Status {get;}
ToString	ScriptMethod	System.Object ToString();

Figure 12-13

Use caution when using the **-PSDisableSerialization** parameter, because although live methods are returned by activities in a workflow when it is specified with a value of **True**, those methods can't be saved if and when a checkpoint occurs in the workflow.



## In Depth 13

# Scope in Windows PowerShell

## Scope in Windows PowerShell

The dictionary definition of “scope” is “extent or range of view, outlook, application, effectiveness, etc.” That’s a good definition for our purposes, because PowerShell’s scope defines the range of effectiveness for a number of elements, including variables, functions, and aliases.

## Types of scope

PowerShell starts with one top-level scope: the global scope. This is the only scope that is not contained within any other scope and you can think of it as the ultimate parent, or root, scope. The command line itself exists within the global scope, and any variables, PSDrives, aliases, or functions that you define interactively all exist in the global scope.

Whenever you run a script, PowerShell creates a new scope to contain the script. This script scope is a child of the scope that ran the script. In other words, if you run a script in the global scope, the new script scope is a child of the global scope. If one script runs another, then the second script’s scope is a child of the first script’s scope, and so forth. It’s essentially a hierarchy, not unlike the file system’s hierarchy of folders and subfolders.

The inside of a function, script block, or filter is a private scope as well, and it is a child of whatever scope contains it. A function within a script has a scope that is a child of the script’s scope—again, similar to the hierarchy of folders on the file system.

## Scope-aware elements

Several elements of PowerShell are scope-aware:

- Variables
- Functions
- Aliases
- PSDrives

Those last two may be surprising, but, in fact, you can define an alias within a script—which has its own scope—and that alias will exist only within that scope. This plays directly into PowerShell's scoping rules, which we'll discuss next.

## Scope rules

PowerShell's rules regarding scopes are simple: when you try to access an element, PowerShell first checks to see if it's defined in the current scope. If it is, then you'll be able to access it for both reading and writing—meaning you'll be able to use the element and change the element, if desired.

If the specified element isn't available in the current scope, then PowerShell starts looking up the hierarchy, starting with the parent of the current scope, then its parent, then its parent, and so forth, up to the top-level global scope. If PowerShell finds the specified element at some point, then you'll have access to use it, but not change it—at least, not without using a special technique. A simple script is probably the easiest way to illustrate this.

```
function Test-Scope {
    Write-Output $example # line 2
    $example = 'Two'      # line 3
    Write-Output $example # line 4
}

$example = 'One'         # line 7
Write-Output $example   # line 8

Test-Scope             # line 10
Write-Output $example   # line 11
```

This script produces the output shown in Figure 13-1.

Figure 13-1

On line 7, the script places the value “One” into the **\$example** variable. Line 8 writes this, resulting in the first line of output. Line 10 then calls the function, which is a new scope. Line 2 writes the current value of **\$example**. This variable doesn’t exist in the function’s scope, so PowerShell looks up one scope to the function’s parent, which is the script itself. PowerShell finds the **\$example** variable, and then line 2 outputs it—resulting in our second line of output. Line 3 attempts to change the value of the **\$example** variable. However, by default, a scope cannot change elements from its parent. Therefore, PowerShell creates a new **\$example** variable within the current scope. Line 4 attempts to access **\$example**, and now it does exist in the current scope, resulting in our third line of output. With the function complete, we execute line 11. Our last line of output is One, because that’s still the value of **\$example** within the current scope. Parent scopes—the script, in this case—cannot see inside their child scopes; they cannot see the function.

The same thing applies to functions, aliases, and PSDrives.

```
function Test-Scope {
    New-PSDrive -Name Z -PSProvider FileSystem -Root C:\test
    Get-ChildItem Z:
}

Test-Scope
Get-ChildItem Z:
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history is as follows:

- PS C:\> function Test-Scope {
- PS C:\> >> New-PSDrive -Name Z -PSProvider FileSystem -Root C:\test
- PS C:\> >> Get-ChildItem Z:
- PS C:\> >> }
- PS C:\> >>
- PS C:\> Test-Scope
- PS C:\>

Name	Used (GB)	Free (GB)	Provider	Root	CurrentLocation
Z	-----	-----	112.82	FileSystem	C:\test

- PS C:\> Get-ChildItem Z:
- Get-ChildItem : Cannot find drive. A drive with the name 'Z' does not exist.
- At line:1 char:1
- + Get-ChildItem Z:
- + CategoryInfo : ObjectNotFound: (Z:String) [Get-ChildItem], DriveNotFoundException
- + FullyQualifiedErrorId : DriveNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
- PS C:\> \_

Figure 13-2

Try running the command in Figure 13- 2 (assuming you have a folder named C:\Test). The function maps a new drive, Z:, to the “C:\Test” folder, and returns a directory listing. After the function exits, the script tries to get a directory listing of Z: and fails because the Z: mapping only exists inside the function’s scope.

Aliases work the same way.

```
function Test-Scope {
    New-Alias -Name plist -Value Get-Process
    plist -Name 'PowerShell Studio'
}
```

```
Test-Scope
plist -Name 'PowerShell Studio'
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> function Test-Scope { ... } is entered, defining a function named Test-Scope. Inside the function, New-Alias -Name plist -Value Get-Process is run, creating an alias named plist for the Get-Process cmdlet. The function then ends with a closing brace. Below the function definition, PS C:\> Test-Scope is run, which executes the function. The output shows a table with columns Handles, NPM(K), PM(K), WS(K), VM(M), CPU(s), Id, and ProcessName. One row is present, showing 2084 handles, 513 NPM, 155620 PM, 242544 WS, 868 VM, 4.95 CPU, Id 3564, and ProcessName 'PowerShell Studio'. After this, PS C:\> plist -Name 'PowerShell Studio' is run, which fails because the alias 'plist' is not recognized. The error message is: "plist : The term 'plist' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again." It also shows the stack trace: At line:1 char:1 + plist -Name 'PowerShell Studio' + ~~~~~ + CategoryInfo : ObjectNotFound: (plist:String) [], CommandNotFoundException + FullyQualifiedErrorId : CommandNotFoundException

Figure 13-3

Again, the function defines a new alias, and then uses it. The script is unable to use that alias because the alias definition only exists within the function's scope.

## Specifying scope

When you look at these four elements—functions, aliases, PSDrives, and variables—you'll find that three of them have specific cmdlets used to create new elements.

- **New-Alias** is used to create new aliases.
- **New-Variable** is used to create new variables.
- **New-PSDrive** is used to create new PSDrives.

Each of these three cmdlets supports a **-Scope** parameter, which allows you to create a new element in a scope other than the current one. The **Set-Variable** cmdlet, which allows you to change an existing variable, also has a **-Scope** parameter, which allows you to change the value of a variable in a scope other than the current one.

## By the way...

Other cmdlets used to deal with aliases, variables, and PSDrives also support a **-Scope** parameter, such as **Remove-Alias** and **New-PSDrive**.

All of these scope parameters can accept one of several values:

- **Global** references the global scope.
- **Script** references the first script that is a parent of the current scope.
- **Local** references the current scope.
- **Numbered scopes** with 0 representing the current scope, 1 representing the current scope's parent, and so forth.

As a general rule, it's considered to be against best practices to have a scope modify anything in its parent or parent's scope. That's because as you move around and reuse an element—such as a script or function—in different ways, you can't be sure what the state of the parent scope will be. Modifying a parent scope involves a risk that you'll affect some other process or operation. However, sometimes it's necessary, which is why the **-Scope** parameter exists. For example, here's how to create a function that defines a new alias in the global scope.

```
function Test-Scope {
    New-Alias -Name plist -Value Get-Process -Scope global
}
```

Or, to change the value of a global variable named **\$example**.

```
Set-Variable -Name $example -Value 'New Value' -Scope global
```

Variables provide a shortcut reference, which may be easier to remember.

```
$Global:example = 'New Value'
```

Again, you can use the keywords **Global**, **Local**, and **Script** with this syntax.

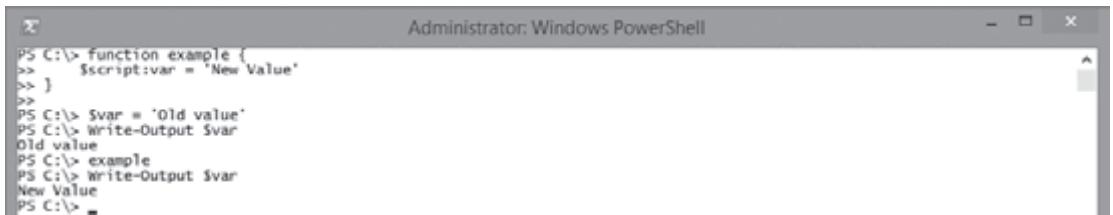
## Best practices for scope

As we've already mentioned, avoid modifying parent scopes, unless there's absolutely no other way to accomplish what you need. Functions are a good example. A function should never do the following:

```
function example {
    $script:var = 'New Value'
}

$var = 'Old value'
Write-Output $var

example
Write-Output $var
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> function example { >> \$var = 'New Value' >> } is entered. Then PS C:\> \$var = 'Old value' is entered, followed by PS C:\> Write-Output \$var which outputs "Old value". Next, PS C:\> example is entered, followed by PS C:\> Write-Output \$var which outputs "New Value".

```
PS C:\> function example {
>>     $var = 'New Value'
>> }
PS C:\> $var = 'Old value'
PS C:\> Write-Output $var
Old value
PS C:\> example
PS C:\> Write-Output $var
New Value
PS C:\> _
```

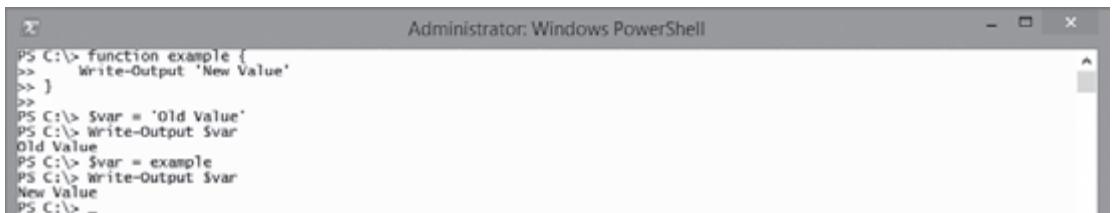
Figure 13-4

Why? Well, for one, if you ever reuse this function in a different script, **\$var** might not be the right variable name. By tying the function to this script, we've limited the function's reusability. Instead, functions should output their return values or collections.

```
function example {
    Write-Output 'New Value'
}

$var = 'Old Value'
Write-Output $var

$var = example
Write-Output $var
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> function example { >> Write-Output 'New Value' >> } is entered. Then PS C:\> \$var = 'Old Value' is entered, followed by PS C:\> Write-Output \$var which outputs "Old Value". Next, PS C:\> \$var = example is entered, followed by PS C:\> Write-Output \$var which outputs "New Value".

```
PS C:\> function example {
>>     Write-Output 'New Value'
>> }
PS C:\> $var = 'Old Value'
PS C:\> Write-Output $var
Old Value
PS C:\> $var = example
PS C:\> Write-Output $var
New Value
PS C:\> _
```

Figure 13-5

This way, you can easily drop the function into any script, or even the global shell, and use it safely.

Our second recommendation is to always assign a value to a variable before using it in the current scope.

```
function sample {
    $var = $input1 + $input2
    Write-Output $var
}
```

What will the function return? Well, we have no idea—it depends on what the **\$input1** and **\$input2** variables contain. The script that this function resides in might have defined those variables, or it might not have. Instead, use variables only after they've been explicitly assigned a value in the current scope.

```
function sample {
    $input1 = 1
    $input2 = 2
    $var = $input1 + $input2
    Write-Output $var
}
```

Or, in the case of a function, define them as input arguments with default values.

```
function sample ($input1 = 1, $input2 = 2) {
    $var = $input1 + $input2
    Write-Output $var
}
```

This ensures that the function won't pick up a variable from a parent scope, by accident. Another way to look at this best practice is that functions should never rely on a variable from a parent scope. If you need to use information inside of a function, pass it into the function via arguments. So, the only way data is included in a function is via arguments, and the only way data is excluded from a function is by being returned from the function. This makes functions self-contained and self-reliant, and you won't need to worry about their parent scope.

## Forcing best practices

PowerShell comes with a cmdlet called **Set-Strict**, which is designed to enforce specific best practices. When you violate those best practices in a given scope, the shell will return an error. When running the **Set-Strict** cmdlet, it only affects the scope in which it is run, as well as any child scopes that do not run **Set-Strict** on their own.

First, you should know how to turn off **Strict** mode.

```
Set-StrictMode -off
```

When you turn on strict mode, you have two choices: **Version 1** or **Version 2**.

```
Set-StrictMode -version 1
```

**Version 1** will throw an exception if you use a variable in the current scope without first assigning a value to that variable within the current scope. In other words, it prevents the current scope from searching its parents for the variable.

```
Set-StrictMode -version 2
```

**Version 2** will also throw an exception for uninitialized variables. In addition, if you attempt to refer to nonexistent properties of an object, or try to use certain invalid forms of syntax (such as calling functions using parentheses and commas with the parameters), **Version 2** will throw an exception. With strict mode off, these operations are allowed, but will typically return unexpected results. For example, you have a function named **MyFunction**, which accepts two input parameters. With **Strict** mode off, the following example is legal syntax.

```
$result = MyFunction(1, 2)
```

This syntax is something you'd find in a language like VBScript, where it works fine. Yet in PowerShell, it passes an array of two elements to the function's first input parameter, while leaving the second parameter blank. This usually isn't what was intended, and so **Strict** mode prohibits this syntax entirely, preventing the unexpected behavior which would otherwise result.

## Dot sourcing

Dot sourcing is a clever technique that tells PowerShell to execute something—usually a script—in the current scope, rather than creating a new scope. For example, consider the following script, which we've named “sample.ps1”.

```
New-Alias -Name PList -Value Get-Process
```

This simple script defines a new alias. Run the following script.

```
.\sample.ps1
PList
```

Figure 13-6

PowerShell executes the script, which defines the **PList** alias, and then discards the script scope when the script ends, so the **PList** alias no longer exists. However, Figure 13-7 shows what happens if you dot source the script.

```
. .\sample.ps1
PList
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\Scripts> . .\sample.ps1 PList has been run. The output is a table of process statistics:

Handles	NPM (K)	PM (K)	WS (K)	VM (M)	CPU(s)	Id	ProcessName
57	7	3752	8540	58	0.41	2244	conhost
353	11	2040	3884	48	0.31	492	csrss
90	9	1376	3252	43	0.08	564	csrss
175	12	1808	20608	126	1.13	3048	csrss
211	16	22512	25068	98	0.08	968	dum
222	23	13148	47020	140	0.25	2156	dum
1255	90	40288	72336	448	3.20	2068	explorer
0	0	0	24	0	0	0	Idle
360	26	9492	26992	248	0.16	864	LogonUI
911	28	5088	10716	42	1.31	656	tsass
502	76	74012	40236	273	52.02	1440	MsMpEng
244	187	3760	1240	45	0.09	4092	NiSrv
77	7	1236	8124	90	0.11	2560	notepad
464	28	54648	63684	626	1.97	2252	powershell
2009	494	152548	239132	849	4.97	3564	PowerShell Studio
697	58	108288	141732	904	15.27	1636	powershell_ise
216	10	3604	6628	82	0.17	2440	rdpclip
103	7	1092	4188	32	0.00	1600	SearchFilterHost
602	35	15668	13668	265	0.80	2892	SearchIndexer
260	9	1416	4820	36	0.00	3956	SearchProtocolHost
218	10	3776	7368	34	0.55	648	services
47	2	280	1040	4	0.09	400	smss

Figure 13-7

Now the script runs without creating a new scope. Since it was run from the shell, which is the global scope, the script's commands all execute within the global scope. Now the **PList** alias will remain defined after the script runs, because we executed the **New-Alias** cmdlet within the global scope.

Dot sourcing can be a useful technique for including a script library file. For example, suppose you have a script file named Library.ps1, which contains a function named **Ping-Computer**. You can easily reuse that across multiple scripts. Here's an example of a script that includes Library.ps1.

```
# include library functions
. .\library.ps1

# use library function
Ping-Computer localhost
```

The script shown dot sources Library.ps1, so Library.ps1 executes in this script's scope, rather than launching in a new child scope. Therefore, everything defined in Library.ps1, including the **Ping-Computer** function, is now defined inside this script's scope, making those functions available for use. PowerShell doesn't include a dedicated **Include** statement simply because dot sourcing provides that functionality already.

## By the way...

PowerShell follows its execution policy when dot sourcing scripts. For example, if we had located Library.ps1 on a network folder and accessed it via a UNC path, then Library.ps1 would have to be digitally signed and the execution policy in PowerShell would need to be set to **RemoteSigned**. If both of those conditions weren't true, PowerShell would then pause the script execution and prompt you for permission to run Library.ps1.

## Nested prompts

Earlier, we introduced you to the concept of nested prompts. A nested prompt exists in the same scope as it was created.

```
function Test1 {
    $var = 3
    Function Test2 {
        $var = 5
        $host.EnterNestedPrompt()
    }
    Test2
}
$var = 1
Test1
```

When we run this, the script sets the **\$var** variable to 1, and calls the **Test1** function. It then sets **\$var** to 3, creating a new **\$var** in this scope, and calls the **Test2** function. **Test2** sets **\$var** to 5, creating yet another version of **\$var** in the local scope. **Test2** then opens a nested prompt. If, within that nested prompt, we examined the value of **\$var**, we would find it to be 5.

```
PS C:\test> .\demol.ps1
PS C:\test>>> $var
5
PS C:\test>>> exit
PS C:\test>
```

Figure 13-8

This behavior ensures that nested prompts remain a useful debugging technique. The nested prompt exists within the scope in which the prompt is called, so it has access to all of that scope's PSDrives, variables, aliases, functions, and so forth.

## Tracing complicated nested scopes

Dealing with scopes can become complicated when they're very deeply nested—in fact, that complication is just one more reason to observe our two main best practices: don't use one scope to mess with another and don't use variables without assigning them a value inside the current scope. Consider the following example (the script Library.ps1, referred to in the example, contains a function named “four”).

```
function One {
    $var1 = "One"
    function Two {
        $var2 = "Two"
        function Three {
            $var3 = "Three"
            .\ExternalScript.ps1
            ..\Library.ps1
        }
        $host.EnterNestedPrompt()
    }
}
$var = "Zero"
```

Here are some statements regarding this code:

- The **\$var** variable exists in the script scope, which is a child of the global scope.
- The **\$var2** variable exists in the scope of function Two, which is a child of function One, which is a child of the script, which is a child of the global scope—that means **\$var2** is four levels deep.
- Function Four, which the Library.ps1 script defines, exists five levels deep. We dot sourced it into function Three. Inside function Four is a new scope, which is six levels deep.
- **ExternalScript** is running in its own scope, because it wasn't dot sourced. The script scope for **ExternalScript** is six levels deep: It's a child of function Three, which is a child of function Two, which is a child of function One, which is a child of the script, which is a child of the global scope.
- Function Two creates a nested prompt, which is in the same scope—four levels deep—as function Two itself.

PowerShell doesn't provide any built-in means of determining how deeply nested the current scope is, so you have to pay close attention to scopes as you're writing and working with scripts. You have to remember the rules for scope use and dot sourcing, and keep track of these things—on a piece of paper, if necessary!

## Special scope modifiers

The **Using** scope modifier, which was introduced in PowerShell version 3, allows local variables to be used in remote sessions.

```
$var = "C:\"
Invoke-Command -ComputerName dc01 {
    Get-ChildItem $using:var
}
```

The screenshot shows an Administrator Windows PowerShell window. The command entered was:

```
PS C:\> $var = "C:\"
PS C:\> Invoke-Command -ComputerName dc01 {
>>>     Get-ChildItem $using:var
>>>
>>>
```

The output shows the contents of the C:\ directory on the remote computer dc01:

Mode	LastWriteTime	Length	Name	PSComputerName
d----	8/22/2013 10:52 AM		PerfLogs	dc01
d-r--	8/22/2013 9:50 AM		Program Files	dc01
d----	8/22/2013 10:39 AM		Program Files (x86)	dc01
d----	10/3/2013 5:17 AM		share	dc01
d-r--	9/20/2013 3:43 PM		Users	dc01
d----	9/20/2013 3:57 PM		Windows	dc01

Figure 13-9

The **Using** scope modifier has some special significance in workflows, which will be covered in addition to the workflow (**\$Workflow:**) scope modifier in the In Depth Chapter 16 “Windows PowerShell Workflow.”

## In Depth 14

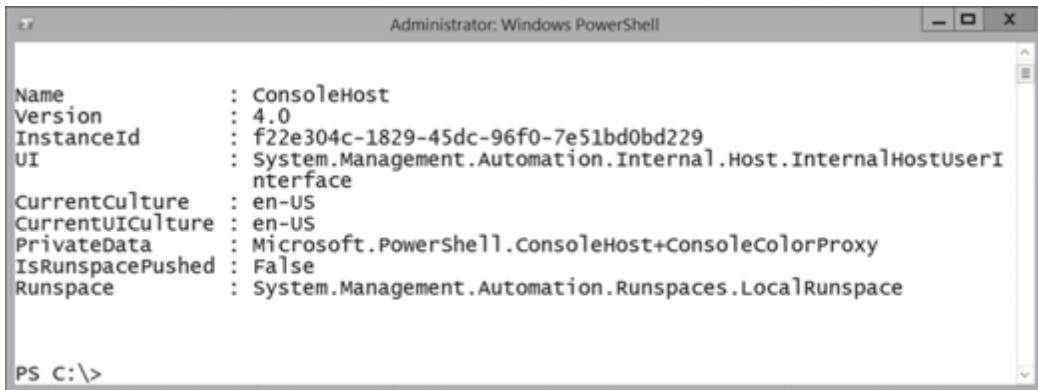
# Working with the PowerShell host

## Working with the PowerShell host

Whenever we're using Windows PowerShell interactively, we're working with what's called the PowerShell hosting application, or just PowerShell host. PowerShell's engine can be hosted by many different applications, such as Exchange Server 2010's management GUI, applications like SAPIEN PrimalScript and PowerShell Studio, and more. When using the PowerShell.exe console host, the special variable **\$host** provides access to some of this host's unique capabilities.

It is important that you understand the **\$host** variable will vary depending on which application is hosting PowerShell. For many of you, the PowerShell that you downloaded from Microsoft may be the only host you ever work with. To illustrate the concept of different hosts, let's look at the different values that exist for each host. Here is **\$host** variable from a PowerShell session running on Windows 8.

```
$host
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> is at the bottom. The output shows host properties:

```

Name      : ConsoleHost
Version   : 4.0
InstanceId: f22e304c-1829-45dc-96f0-7e51bd0bd229
UI        : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture: en-US
CurrentUICulture: en-US
PrivateData  : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
IsRunspacePushed: False
Runspace    : System.Management.Automation.Runspaces.LocalRunspace

```

Figure 14-1

We've mentioned that PrimalScript and PowerShell Studio can host PowerShell. Here is the information for the host in PowerShell Studio.

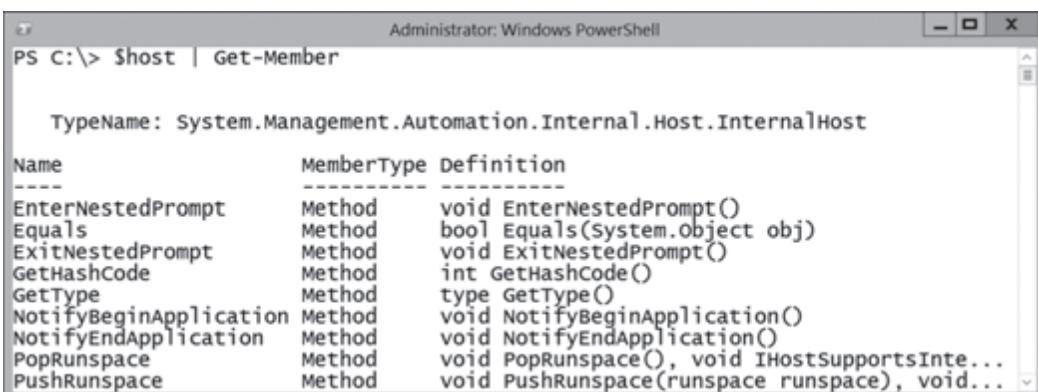
```

Name      : PrimalScriptHostImplementation
Version   : 3.0.0.0
InstanceId: 472f058e-69c4-4c85-b193-e355deed7eab
UI        : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture: en-US
CurrentUICulture: en-US
PrivateData  :
IsRunspacePushed:
Runspace    :

```

For the most part, these hosts have nearly identical functionality, but that's not to say that some other future host might have additional functionality. One way to check what your host can do is to pipe **\$host** to **Get-Member**.

```
$host | Get-Member
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$host | Get-Member is at the bottom. The output shows the members of the InternalHost type:

Name	MemberType	Definition
EnterNestedPrompt	Method	void EnterNestedPrompt()
Equals	Method	bool Equals(System.Object obj)
ExitNestedPrompt	Method	void ExitNestedPrompt()
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
NotifyBeginApplication	Method	void NotifyBeginApplication()
NotifyEndApplication	Method	void NotifyEndApplication()
PopRunspace	Method	void PopRunspace(), void IHostSupportsInte...
PushRunspace	Method	void PushRunspace(runspace runspace), void...

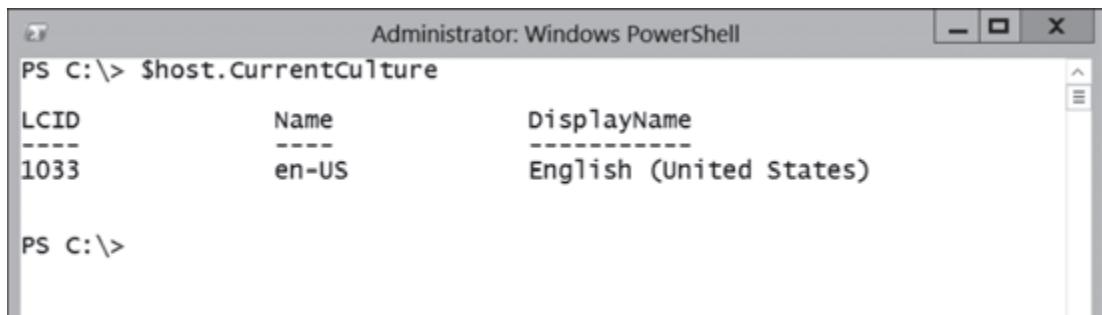
Figure 14-2

You see most of the properties when you invoke **\$host**. But what else is there?

## Culture clash

Given that Windows is an international platform, it should come as no surprise that different versions have different regional and language settings. In PowerShell, this is referred to as the Culture, which is a property of **\$host**.

```
$host.CurrentCulture
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$host.CurrentCulture is run, resulting in the following table:

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

PS C:\>

Figure 14-3

You receive the same result if you use the **Get-Culture** cmdlet. However, there may be situations where you need to execute a command or expression in another culture setting. Usually, the situation is with non-US users who are running a command or application that fails to execute properly unless they use the EN-US culture. Even though PowerShell has a cmdlet to retrieve the current culture, there are no cmdlets for changing it system wide, which isn't very practical anyway. Culture settings are thread level. But what if you really, really had to change the culture setting temporarily? The PowerShell team at Microsoft posted a function a while ago on their blog.

```

1 Function Using-Culture {
2     Param (
3         [System.Globalization.CultureInfo]$culture = `n
4             '$(throw "USAGE: Using-Culture -Culture culture -Script {scriptblock}"),`n
5         [String]$script=$(throw "USAGE: Using-Culture -Culture culture -Script {scriptblock}")`n
6     )
7     $OldCulture = [System.Threading.Thread]::CurrentThread.CurrentCulture
8     trap
9     {
10         [System.Threading.Thread]::CurrentThread.CurrentCulture = $OldCulture
11     }
12     [System.Threading.Thread]::CurrentThread.CurrentCulture = $culture
13     Invoke-Expression $script
14     [System.Threading.Thread]::CurrentThread.CurrentCulture = $OldCulture
15 }

```

Figure 14-4

Using this function, you can execute any expression or command under the guise of a different culture.

```
Using-Culture en-GB {Get-Date}
```

The function executes **Get-Date** using the British culture settings, which have a different date format than the United States settings.

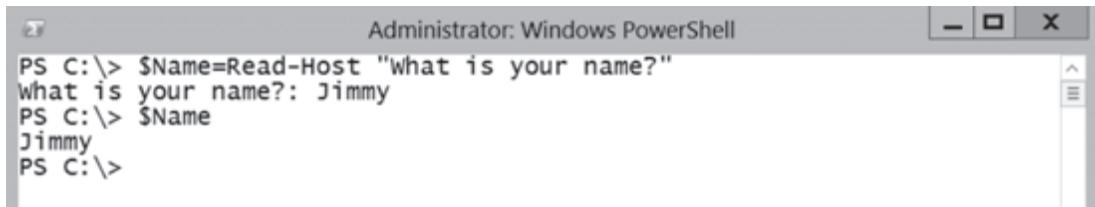
## Using the UI and RawUI

The **\$host** object also gives you access to the underlying user interface (UI). You shouldn't need to access these properties and methods often, and there are cmdlets for much of the functionality you are likely to need. Still, there may be situations where you'd like to tweak the PowerShell window or take advantage of a feature that doesn't have a ready cmdlet.

## Reading lines and keys

The best way to see user input is by using the **Read-Host** cmdlet.

```
$name=Read-Host "What is your name?"`n$name
```



```
Administrator: Windows PowerShell
PS C:\> $Name=Read-Host "What is your name?"
what is your name?: Jimmy
PS C:\> $Name
Jimmy
PS C:\>
```

Figure 14-5

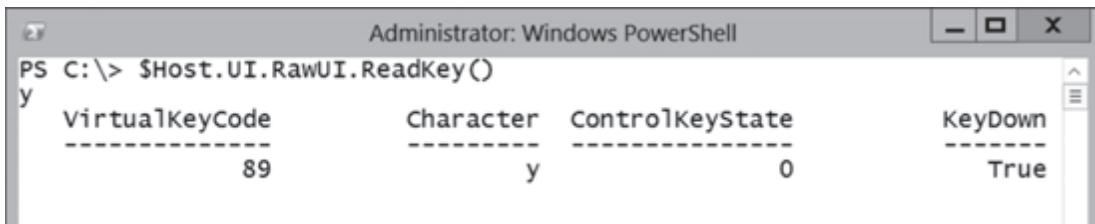
You can use the **ReadLine()** method from **\$host.UI**, but it's a little primitive.

```
Write-Host "What is your name?";$name=$host.UI.ReadLine()
```

The **ReadLine()** method has no provision for a prompt like **Read-Host**, so we used **Write-Host** to display something. The **ReadLine()** method simply waits for the user to type a line and press **Enter**. Offhand, we can't think of a situation where this would be preferable to using **Read-Host**. However, there is another **\$host** method that you might find helpful, for which there is no PowerShell cmdlet.

The **\$host.UI.RawUI** object has a method called **.ReadKey()**, which works like the **ReadLine()** method, except it only accepts a single key.

```
$Host.UI.RawUI.ReadKey()
y
```



```
Administrator: Windows PowerShell
PS C:\> $Host.UI.RawUI.ReadKey()
y
+-----+-----+-----+-----+
| VirtualKeyCode | Character | ControlKeyState | KeyDown |
+-----+-----+-----+-----+
|          89      |      y     |          0       |   True  |
```

Figure 14-6

After typing the command, PowerShell will wait for you to press any key. When you do, it displays the key. You can fine-tune this method by specifying some **.ReadKey()** options. You'll likely not need to echo the typed character, so you can specify that option, along with a few others.

## Changing the window title

You can access the title of your Windows PowerShell window very easily, by accessing the **WindowTitle** property.

```
$host.UI.RawUI.WindowTitle
```

You can just as easily change the window title.

### Create-ServerReport.ps1

```
function Set-Title {
    Param (
        [string]$NewTitle
    )
    $host.UI.RawUI.WindowTitle=$NewTitle
}
Set-Alias title Set-Title

function Save-Title {
    $script:SavedTitle=$host.UI.RawUI.WindowTitle
}

#call the Save-Title function
Save-Title
$report="report.txt"

#write a line to the report
"REPORT CREATED $(Get-Date)" | Out-File $report

Get-Content servers.txt | foreach {
    #skip blank lines
    if (( $_ ).length -gt 0)
    {
        #create a new variable to make things easier to read and
        #trim off any extra spaces. Make the name upper case
        #while we're at it.
        $server = ($_.Trim()).ToUpper()

        $newtitle="Checking $server"
        #set a new title
        title $newtitle

        $server | Out-File $report -Append

        "Operating System" | Out-File $report -Append
        Get-WmiObject win32_operatingsystem -Computer $server | Out-File $report -Append

        "ComputerSystem" | Out-File $report -append
        Get-WmiObject win32_computersystem -Computer $server | Out-File $report -Append

        #pause for a few seconds just to show title in action
        #not really required.
        sleep 2
    } #end If
} #end foreach
```

```
#revert back to old title
Title $script:savedtitle

#view report
Notepad $report
```

This script processes a list of servers, retrieves information about each server from WMI, and then saves the results to a text file. Since this script could run for a long time, we modify the title to reflect what server the script is working on. You can then minimize your PowerShell window, yet still be able to monitor the script's progress.

The script defines functions to set the window title, as well as save the current one. We also define an alias, or **Title**, for the **Set-Title** function. Thus as each server is processed from the list, the window title is changed to reflect the current server.

```
$newtitle="Checking $server"
Title $newtitle
```

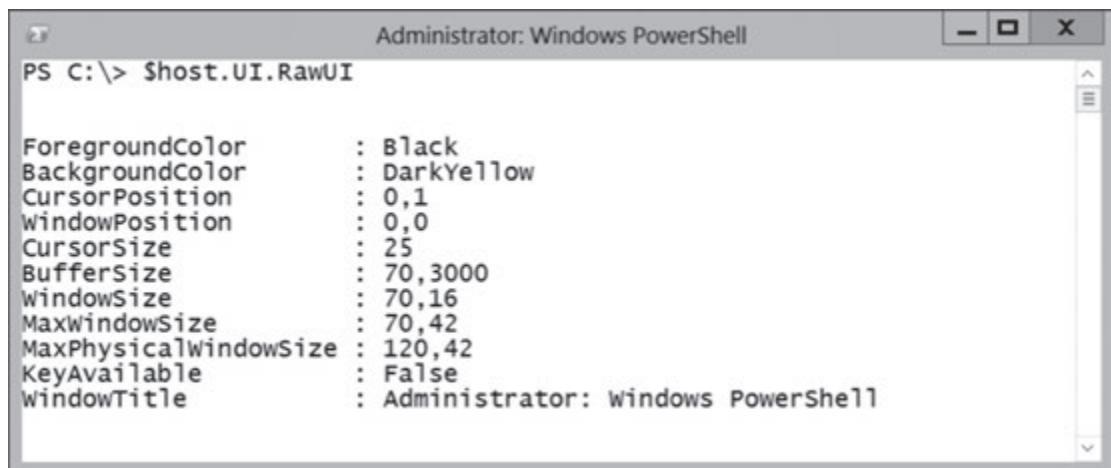
At the end of the script, we change the title back to the saved, original window title.

```
Title $script:savedtitle
```

## Changing colors

We can also modify the color settings of PowerShell windows. First, let's look at the current settings.

```
$host.UI.RawUI
```



A screenshot of an Administrator: Windows PowerShell window. The command PS C:\> \$host.UI.RawUI is entered at the prompt. The output shows various properties of the RawUI object:

ForegroundColor	:	Black
BackgroundColor	:	DarkYellow
CursorPosition	:	0,1
WindowPosition	:	0,0
CursorSize	:	25
BufferSize	:	70,3000
WindowSize	:	70,16
MaxWindowSize	:	70,42
MaxPhysicalWindowSize	:	120,42
KeyAvailable	:	False
WindowTitle	:	Administrator: Windows PowerShell

Figure 14-7

You might prefer something like the following:

```
$host.UI.RawUI.BackgroundColor = "green"
$host.UI.RawUI.ForegroundColor = "black"
```

These changes last only for as long as your PowerShell session is running. If you want a more permanent change, then you will need to add lines like these to your profile script.

## Changing window size and buffer

The raw UI also lets you control position and size of your PowerShell windows. Figure 14-8 shows an expression that allows you to see the current window size.

```
$Host.UI.RawUI.WindowSize | Format-List
```

```
Administrator: Windows PowerShell
PS C:\> $Host.UI.RawUI.WindowSize | Format-List

Width : 70
Height : 16
```

Figure 14-8

The **WindowSize** cannot be larger than the value of **MaxPhysicalWindowSize**. Here's a function that simplifies changing the console window size.

```
function Set-WindowSize {
    param(
        [int]$width=$host.UI.RawUI.WindowSize.width,
        [int]$height=$host.UI.RawUI.WindowSize.height
    )
    $size = New-Object System.Management.Automation.Host.Size ($width, $height)
    $host.UI.RawUI.WindowSize=$size
}
```

Once you've loaded the function, you can use the **Set-WindowSize** command to dynamically change your console window to 60 columns wide by 30 rows high.

```
Set-WindowSize 60 30.
```

If you don't specify a width or height value, the function will use the value of the current window size. We can take a similar approach to the console's buffer.

The buffer controls how much of the window can be scrolled either vertically or horizontally. Setting a large vertical buffer lets you see more output from previous commands. If you have a script that will produce a lot of information, you may want to modify the buffer size so that you can scroll up to see it all. Here's a function that might help.

```
function Set-WindowBuffer {
    param(
        [int]$width=$host.UI.RawUI.BufferSize.width,
        [int]$height=$host.UI.RawUI.BufferSize.height
    )
    $buffer = New-Object System.Management.Automation.Host.Size ($width, $height)
    $host.UI.RawUI.BufferSize = $buffer
}
```

You cannot set a buffer size that is less than the window size.

```
Set-WindowBuffer 120 500
```

If you run the previous command, when your current window and buffer size are 120 x 50, you'll see a vertical scroll bar appear in your PowerShell window. Now, you can scroll back to see more of your previous commands and output.

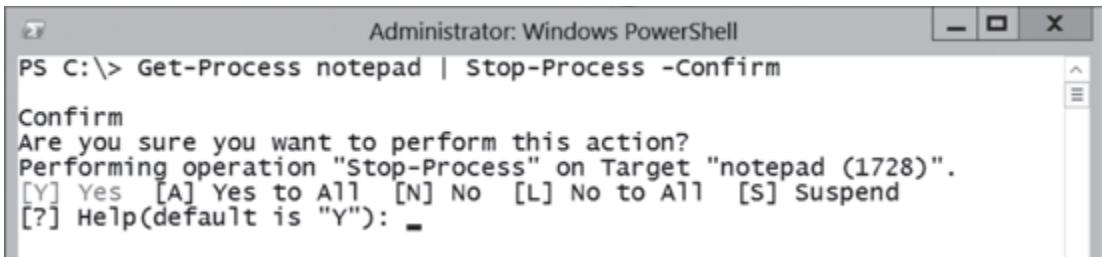
## Nested prompts

One of the most interesting features about the console host is its nested prompts. This is a bit difficult to see in action, so let's walk through an example. Be very careful if you're following along! One wrong keystroke could crash your computer.

First, open several instances of Windows Notepad. Then, in PowerShell, run the following:

```
Get-Process notepad | Stop-Process -Confirm
```

You should see something like the following:



A screenshot of the Windows PowerShell interface. The title bar says "Administrator: Windows PowerShell". In the command line, the user has run the command "Get-Process notepad | Stop-Process -Confirm". A "Confirm" dialog box is displayed, asking if the user wants to perform the action. It shows the target process "notepad (1728)". The user has typed "[?] Help(default is \"Y\"):" followed by a space. The dialog box has standard window controls (minimize, maximize, close) and scroll bars.

```
Administrator: Windows PowerShell
PS C:\> Get-Process notepad | Stop-Process -Confirm
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (1728)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend
[?] Help(default is "Y"):
```

Figure 14-9

If you press **S**, you'll suspend the pipeline operation and enter a nested prompt. Your original command is still pending, but you've sort of entered a side conversation with Windows PowerShell. In it, you can do whatever you want—check variables, run cmdlets, and so forth. By default, the PowerShell prompt reflects this nested state.

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

Typing **Exit** ends the nested prompt and takes you back up one level.

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

Here, you'll see your original command still in action. Select **L** to abort. You can manually create a new nested prompt by running **\$host.EnterNestedPrompt()**. There's not much use in this from the command line, perhaps, but when you're writing a script you can have a new prompt created for certain conditions, such as an error occurring. The nested prompt runs inside the script, so it'll have access to all the variables and so forth within a script. Again, running **Exit** ends the nested prompt and would return to your script.

Note that you can have more than a few nested prompts (we can't imagine why you'd need even that many, to be honest). A built-in variable, **\$NestedPromptLevel**, tells you how deeply you're already nested and you can check it to see if it's safe to launch a new nested prompt.

## By the way...

Instead of running **Exit**, you can also run **\$host.ExitNestedPrompt()** to exit a nested prompt.

## Quitting PowerShell

Of course, running **Exit** from a non-nested prompt will exit the shell. You can also run **\$host.SetShouldExit(xxx)** to exit the shell, passing a number for **xxx**. This number will be returned as an error code to the Windows environment. This could be useful if you are running a PowerShell script wrapped up in a batch file or VBScript.

## Prompting the user to make a choice

The **\$host** interface provides some access to PowerShell's underlying user interface capabilities. For example, suppose you want to provide a simple "Yes or No" text prompt. First, you would need to construct the prompt in an array.

```
$no = ([System.Management.Automation.Host.ChoiceDescription]"&No")
$no.HelpMessage = "Do not continue"
$yes = ([System.Management.Automation.Host.ChoiceDescription]"&Yes")
$yes.HelpMessage = "Continue"
$prompts = ($yes, $no)
```

What we've done is created two new prompts. Use the **&** character before whichever letter will be the answer for that prompt. In other words, in our example, you'd press "N" for "No" and "Y" for "Yes." We also specified a "help message" for each prompt, and then assembled them into an array in the **\$prompts** variable. Next, you ask the host to display the prompt.

```
$host.UI.PromptForChoice("Continue?", "Do you want to continue?", ` 
[System.Management.Automation.Host.ChoiceDescription[]]$prompts, 0)

Continue?
Do you want to continue?
[Y] Yes [N] No [?] Help (default is "Y"):
```

Notice that the prompt does not display our help message text—that's only displayed if you select ? from the prompt. Also, as you can see, you can provide a title, a description, and then your array of choices. Those choices have to be of a specific .NET Framework type, **System.Management.Automation.Host.ChoiceDescription**, which is why you see that long class name enclosed in square brackets above. The **PromptForChoice()** method will return whatever choice the user made. This is perhaps best wrapped up into a reusable function. Here's a sample script.

```
function Set-ChoicePrompt {
    param (
        [string]$Caption = "Continue?",
        [string]$Message = "What do you want to do?",
        [System.Management.Automation.Host.ChoiceDescription[]]$Choices,
        [int]$Default=0
    )

    $host.UI.PromptForChoice ($Caption, $Message, $Choices, $Default)
}
```

```
#define some variables
$caption="Continue with this operation?"
$message="This is serious stuff. Is your resume up to date? Do you want to continue?"
$yes = ([System.Management.Automation.Host.ChoiceDescription]"&Yes")
$yes.HelpMessage = "Continue, I am ready."

$no = ([System.Management.Automation.Host.ChoiceDescription]"&No")
$no.HelpMessage = "Do not continue. Get me out of here."
$choices = ($yes, $no)

#show the prompt and set the default to No
[int]$r=Set-ChoicePrompt -Caption $caption -Message $message -Choices $choices -Default 1

if ($r -eq 0) {
    Write-Host "Good Luck" -ForegroundColor Green
}
else {
    Write-Host "Aborting the mission" -ForegroundColor Yellow
}
```

The script has a relatively simple function: to create the choice prompt. All you have to do is assign the values for all the necessary components. In the body of the sample script, we've set some of these values and invoked the function. The value returned by the prompt is the index number of the choice. We're setting the default to the prompt with an index value of 1, or the **No** option. In the script, the user is prompted and their choice is saved in the variable, **\$r**.

```
[int]$r = Set-ChoicePrompt -Caption $caption -Message $message -Choices $choices -Default 1
```

Using an **If** statement to evaluate the value, the script takes appropriate action depending on the choice.

```
.\Set-ChoicePrompt.ps1
Continue with this operation?
This is serious stuff. Is your resume up to date? Do you want to continue?
[Y] Yes [N] No [?] Help (default is "N"): n
Aborting the mission
PS C:\>
```

This brings us to the end for now of **\$Host**. There are plenty of things you can do to control PowerShell and **\$Host** is but one of them.

## In Depth 15

# Working with XML documents

## Working with XML documents

Windows PowerShell has a very powerful and intuitive way of allowing you to work with complex XML documents. Essentially, PowerShell adapts the structure of the XML document itself into an object hierarchy, so that working with the XML document becomes as easy as referring to object properties and methods.

## What PowerShell does with XML

When PowerShell converts text into XML, it parses the XML's document hierarchy and constructs a parallel object model. For example, the following XML document named "pets.xml" exists in the "C:\test" directory on our workstation.

```
<Pets>
  <Pet>
    <Breed>Ferret</Breed>
    <Age>3</Age>
    <Name>Patch</Name>
  </Pet>
  <Pet>
    <Breed>Bulldog</Breed>
    <Age>5</Age>
    <Name>Chesty</Name>
  </Pet>
</Pets>
```

You can load that XML document into PowerShell.

```
[xml]$xml = Get-Content -Path C:\test\pets.xml
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `PS C:\> [xml]\$xml = Get-Content -Path C:\test\pets.xml` is entered at the prompt. The output shows the command being run and then a blank line.

Figure 15-1

PowerShell then constructs the object hierarchy. For example, you could access the breed of the first pet.

```
$xml.Pets.Pet[0].Breed
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `PS C:\> \$xml.Pets.Pet[0].Breed` is entered at the prompt. The output shows the result as "Ferret".

Figure 15-2

The `$xml` variable represents the XML document itself. From there, you simply specify the document elements as properties: the top-level `<Pets>` element, the first `<Pet>` element (indicated by `pet[0]`, just like an array), and then the `<Breed>` element. If you don't specify a subelement, PowerShell treats sub-elements as properties. For example, here is how to view all of the properties—subelements, that is—of the second pet.

```
$xml.Pets.Pet[1]
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `PS C:\> \$xml.Pets.Pet[1]` is entered at the prompt. The output shows a table with three columns: Breed (Bulldog), Age (5), and Name (Chesty).

Breed	Age	Name
Bulldog	5	Chesty

Figure 15-3

This is a pretty creative way of working with XML and it doesn't require you to use any of the more complex mechanisms that you usually have to deal with when working with XML. Let's move on to a slightly more complex example.

## Basic XML manipulation

As an example, we went to our own blog's RSS feed—RSS just being an XML application, after all—located at <http://www.sapien.com/blog/feed/>, and saved the RSS XML as a local file so that we could work with it.

```
Invoke-WebRequest -Uri "http://www.sapien.com/blog/feed/" -OutFile "C:\test\sapienblog.xml"
```



Figure 15-4

The **Invoke-WebRequest** cmdlet that we used in Figure 15-4 to save our file is a cmdlet that was first introduced in PowerShell version 3.

The following page includes an excerpt; the remainder of the file just has additional **<Item>** nodes containing more blog entries. With this XML in a local file, our first step is to load this into PowerShell and have it recognized as XML.

```
[xml]$rss = Get-Content -Path C:\test\sapienblog.xml
```



Figure 15-5

Simple enough. By specifically casting **\$rss** as an **[xml]** type, we've let PowerShell know that some XML is coming its way. The **Get-Content** cmdlet loads the text from the file, and then PowerShell does the rest. You can view the file in SAPIEN PrimalScript to see what it looks like in a nicely formatted GUI interface.

## Windows PowerShell: TFM

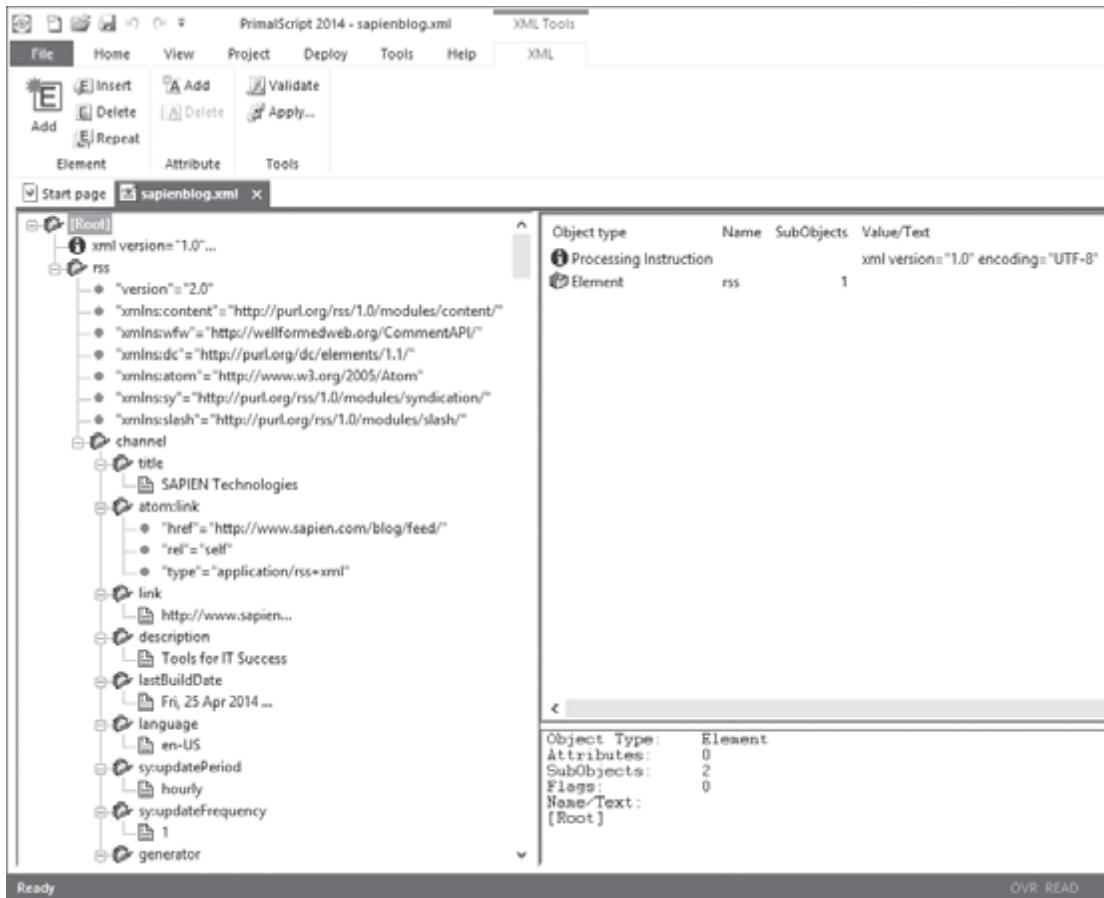


Figure 15-6

What is \$rss?

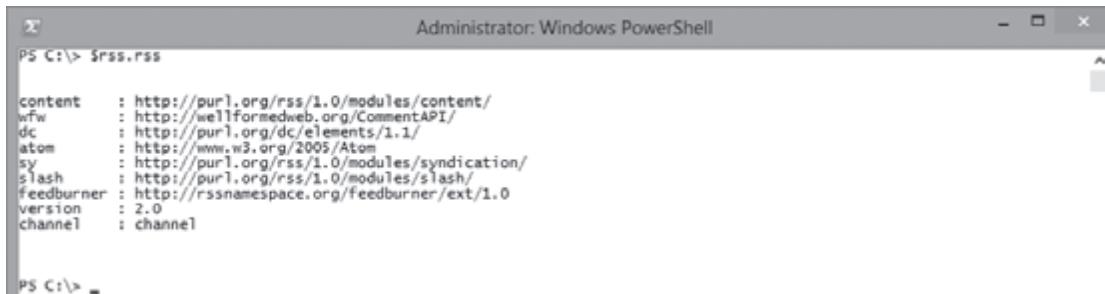
\$rss

```
Administrator: Windows PowerShell
PS C:\> $rss
xml
version="1.0" encoding="UTF-8"
rss
---
```

Figure 15-7

If you have the file opened in a text editor, you'll realize these two properties are the first set of tags. Let's drill down.

\$rss.rss



```
Administrator: Windows PowerShell
PS C:\> $rss.rss
content  : http://purl.org/rss/1.0/modules/content/
wfw      : http://wellformedweb.org/CommentAPI/
dc       : https://purl.org/dc/elements/1.1/
atom     : https://www.w3.org/2005/Atom
sy       : http://purl.org/rss/1.0/modules/syndication/
slash    : http://purl.org/rss/1.0/modules/slash/
feedburner: http://rssnamespace.org/feedburner/ext/1.0
version   : 2.0
channel   : channel

PS C:\> =
```

Figure 15-8

What you’re looking at in Figure 15-8 is the `<rss>` element of the `$rss` variable, our XML document, and you’re seeing the attributes of the `<rss>` tag—go back and refer to the XML excerpt, and you’ll see where these values came from. We didn’t have to do anything special to access them—PowerShell just knew how.

Underneath the `<rss>` tag is a `<channel>` tag, and underneath that is a `<title>` tag. Here’s how we can access the feed’s title.

```
$rss.rss.channel.title
```



```
Administrator: Windows PowerShell
PS C:\> $rss.rss.channel.title
SAPIEN Technologies
PS C:\>
```

Figure 15-9

In other words, the object hierarchy—`rss.channel.title`—mirrors the hierarchy of tags in the XML document. Underneath the `<channel>` tag, we’ll also find multiple `<item>` tags, each one representing a blog posting. Each `<item>` tag has various subtags, including a `<title>` tag, which is the title of that blog posting. Since PowerShell will find more than one `<item>` section, it will create a collection out of them. So, here’s how to access the title of the first blog post.

```
$rss.rss.channel.item[0].title
```



```
Administrator: Windows PowerShell
PS C:\> $rss.rss.channel.item[0].title
Onsite PowerShell Training Offered worldwide
PS C:\>
```

Figure 15-10

What if we want to see the post titles for the first six entries? All we need to do is use the PowerShell range operator [0..5] and select the **Title** property for each item object.

```
$rss.rss.channel.item[0..5] | Select-Object -Property Title
```

```
Administrator: Windows PowerShell
PS C:\> $rss.rss.channel.item[0..5] | Select-Object -Property Title
title
-----
PowerShell.org: New tools in my toolbox
PowerShell Studio 2014: What's New?
Managing Your Subscriptions
PrimalScript 2014: Introducing Restore Points
The 2014Cs Are Here!
SAPIEN World Tour, VersionRecall

PS C:\>
```

Figure 15-11

For the sake of illustration, perhaps we want to change the title of the second post.

```
$rss.rss.channel.item[1].title = "Alternate title"
```

```
Administrator: Windows PowerShell
PS C:\> $rss.rss.channel.item[1].title = "Alternate title"
PS C:\>
```

Figure 15-12

We'll invoke the XML object's **Save()** method to write the XML to a file.

```
$rss.Save("C:\test\revised.xml")
```

```
Administrator: Windows PowerShell
PS C:\> $rss.Save("C:\test\revised.xml")
PS C:\>
```

Figure 15-13

Now we'll replace what's in **\$rss** with the content in the revised file and display the title for the first six entries again. Notice, the title for the second entry is now "Alternate Title".

```
[xml]$rss = Get-Content -Path C:\test\revised.xml
$rss.rss.channel.Item[0..5] | Select-Object -Property Title
```

The screenshot shows an Administrator Windows PowerShell window. The command PS C:\> [xml]\$rss = Get-Content -Path C:\test\revised.xml is run, followed by PS C:\> \$rss.rss.channel.Item[0..5] | Select-Object -Property Title. The output displays the titles of the first five items in the XML file, which include "PowerShell.org: New tools in my toolbox", "Alternate title", "Managing Your Subscriptions", "PrimalScript 2014: Introducing Restore Points", "The 2014Cs Are Here!", and "SAPIEN World Tour, VersionRecall".

Figure 15-14

As you can see, working with XML in PowerShell is fairly straightforward. Our examples up to this point demonstrate how easily you can work with a basic XML file. More complicated files simply create a deeper object hierarchy—that doesn't really change how things work. It's beyond the scope of this book to look at really complicated XML operations, like **XPath** queries and so forth. However, we hope this quick look at XML has given you an idea of what PowerShell can accomplish and offered some possibilities for parsing XML files that you may have in your environment.

## A practical example

So, what good is all this XML stuff? Let's look at a real-world example—one that will also introduce you to additional XML techniques. We're going to start with a basic XML file that contains computer names. We'll call this `Inventory.xml`.

```
<Computers>
<Computer Name="localhost" />
<Computer Name="DC02" />
<Computer Name="SQL02" />
</Computers>
```

We want to inventory some basic information from these computers (of course, you could add more to your list, if you wanted to), including their Windows build number, service pack version, and the amount of free space on their local disk drives. We want our final result to look something like the following:

```
Get-Content -Path C:\test\inventory-out.xml
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Get-Content -Path C:\test\inventory-out.xml is run, displaying an XML document with three computer entries: "localhost", "DC02", and "SQL02". The XML structure includes Computer, OS, Disks, and Status nodes.

```

PS C:\> Get-Content -Path C:\test\inventory-out.xml
<Computers>
  <Computer Name="localhost">
    <OS BuildNumber="9200" ServicePack="0" />
    <Disks>
      <Disk DeviceID="C:" FreeSpace="104134MB" />
    </Disks>
    <Status>Complete</Status>
  </Computer>
  <Computer Name="DC02">
    <OS BuildNumber="9200" ServicePack="0" />
    <Disks>
      <Disk DeviceID="C:" FreeSpace="122097MB" />
    </Disks>
    <Status>Complete</Status>
  </Computer>
  <Computer Name="SQL02">
    <Status>Unreachable</Status>
  </Computer>
</Computers>
PS C:\>

```

Figure 15-15

**Note:**

The code examples shown in this portion of the chapter are snippets of the XMLInventory.ps1 script near the end of the chapter. We're breaking the larger script down into these smaller snippets so that it's easier to explain and understand the script. These snippets are not meant to be run individually and they may generate errors if you attempt to do so.

Our goal is to build a PowerShell script that is not only capable of retrieving the necessary information, but also capable of putting it into this XML format and saving it all back to disk. We'll start by defining a **Get-Status** function, which we'll use to ensure WMI connectivity to a remote computer. This function simply makes an attempt to query a WMI class from the specified computer; its **-ErrorAction** parameter is set to **SilentlyContinue**, so that in the event of an error, no error message will be shown. The built-in **\$?** variable contains a **True** or **False** value, depending on whether the previous command completed successfully, so we're simply outputting that variable as the result of the function.

```

function Get-Status {
  param(
    [string]$ComputerName
  )
  Get-WmiObject -Class Win32_BIOS -ComputerName $ComputerName -ErrorAction SilentlyContinue |
    Out-Null
  Write-Output $?
}

```

Next, we write out a status message and load our inventory XML file from disk. Notice that we're explicitly declaring the **\$xml** variable as an **[XML]** data type, forcing PowerShell to parse the text file as XML.

```

Write-Output "Beginning inventory..."

# load XML
$xml = Get-Content -Path C:\test\inventory.xml

```

Next, we're going to repeat a large block of code once for each `<computer>` node found in the XML. We start by pulling the `Name` attribute of the `<computer>` tag into the `$name` variable. Be careful here, because XML is case-sensitive. Make sure the attribute you are calling is the same case as in the XML file.

```
for ($i=0; $i -lt $xml.computers.computer.count; $i++) {
    # get computername
    $name = $xml.computers.computer[$i].getattribute("Name")
```

We create a new XML node named `<Status>`. Notice that the main XML document, stored in the `$xml` variable, has the capability of creating new nodes—we're specifically creating an element, which is basically an XML tag. We're then executing our **Get-Status** function, passing it the current computer name to test.

```
# create status node and get status
$statusnode = $xml.CreateNode("element", "Status", "")
$status = Get-Status -ComputerName $name
```

If the status comes back as **False**—that is, not **True**, as indicated by the `-not` operator—we set the `<Status>` node's inner text—the text appearing between `<Status>` and `</Status>`—to **Unreachable**. Otherwise, we set the inner text to **Complete** and continue with the rest of our script.

```
if (-not $status) {
    $statusnode.set_innertext("Unreachable")
}
else {
    $statusnode.set_innertext("Complete")
```

If our status check was successful, we'll query the **Win32\_OperatingSystem** class from the remote computer. We're using the `-Filter` parameter of **Get-WMIObject**, so only the instances of **Win32\_LogicalDisk** where the **DriveType** property is equal to 3, indicating a local disk, are retrieved. When possible, try to filter before the first pipeline, so the query only retrieves the necessary objects and properties for those objects. Also try to filter as much as possible on the computer where the data resides, so that the data that's returned to the computer running the script is minimized (your network admin will thank you).

```
# get OS info
$os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName $name

# get local disks
$disks = Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName $name
```

We'll ask the XML document in the `$xml` variable to create `<OS>` and `<Disks>` elements. We'll continue working with these elements to populate them with inventory data.

```
# create OS node, disks node
$osnode = $xml.CreateNode("element", "OS", "")
$disksnode = $xml.CreateNode("element", "Disks", "")
```

Since we have the operating system build number and service pack information available, we can add those attributes to the **<OS>** element.

```
# append OS attrs to node
$osnode.setAttribute("BuildNumber", $os.buildnumber)
$osnode.setAttribute("ServicePack", $os.servicepackmajorversion)
```

Now we append the complete **<OS>** element to the current **<Computer>** node. Notice that we're piping the output of the **AppendChild()** method to the **Out-Null** cmdlet. That's because **AppendChild()** normally displays the node it just finished appending, and that output looks messy when we run our script, so we're sending the output to the **Out-Null** cmdlet to get rid of it.

```
# append OS node to Computer node
$xml.computers.computer[$i].appendChild($osnode) | Out-Null
```

Now it's time to enumerate through the logical disks we retrieved from WMI. We start by creating a new XML element named **<Disk>**, which will store our device ID and free space information.

```
# go through the logical disks
foreach ($disk in $disks) {
    # create disk node
    $disknode = $xml.CreateNode("element", "Disk", "")
```

Next we create the **DeviceID** attribute on the **<Disk>** node. We also convert the free space to megabytes, rather than bytes, by dividing the **FreeSpace** property by 1 MB. We then cast the **FreeSpace** as an integer, so a whole number is returned and we don't wind up with a decimal value. Finally, we add the letters "MB" to the end of the **FreeSpace** numeric value to provide a unit of measurement in our inventory file. We set the **<Disk>** node's **FreeSpace** attribute equal to our megabyte measurement.

```
#create deviceid and freespace attrs
$disknode.setAttribute("DeviceID", $disk.deviceid)
$freespace = "$($disk.freespace / 1MB -as [int])MB"
$disknode.setAttribute("FreeSpace", $freespace)
```

We're now ready to append the current **<Disk>** node to the overall **<Disks>** node. After completing all of the available logical disks, we append the completed **<Disks>** node to the current **<Computer>** node. Again, we're using the **Out-Null** cmdlet to keep the output from the **AppendChild()** method from displaying.

```

    # append Disk node to Disks node
    $disknode.appendChild($disknode) | Out-Null
}

# append Disks node to Computer node
$xml.computers.computer[$i].appendChild($disknode) | Out-Null
}

```

We've reached the end of our **If/Else** construct, which had checked the result of our **Get-Status** function. We can, therefore, append the **<Status>** node, which will either contain **Complete** or **Unreachable**, to the **<Computer>** node. Again, we're piping the output of the **AppendChild()** method to the **Out-Null** cmdlet in order to suppress the output text.

```

# append status node to Computer node
$xml.computers.computer[$i].appendChild($statusnode) | Out-Null
}

```

At this point, we've reached the end of our original **For** loop. We're ready to delete any existing output file and write our modified XML to a new filename, complete with all the inventory information we've added.

```

# output XML
Remove-Item -Path "C:\test\inventory-out.xml" -ErrorAction SilentlyContinue
$xml.save("C:\test\inventory-out.xml")

Write-Output "...Inventory Complete."

```

You can open and view the saved XML file in Internet Explorer or any other application that knows how to read XML files. Here's the full, final script.

```

XMLInventory.ps1

function Get-Status {
    param(
        [string]$ComputerName
    )
    Get-WmiObject -Class Win32_BIOS -ComputerName $ComputerName -ErrorAction SilentlyContinue |
        Out-Null
    Write-Output $?
}

Write-Output "Beginning inventory..."

# load XML
$xml = Get-Content -Path C:\test\inventory.xml

for ($i=0; $i -lt $xml.computers.computer.count; $i++) {

    # get computername
    $name = $xml.computers.computer[$i].getattribute("Name")

    # create status node and get status

```

Windows PowerShell: TFM

```
$statusnode = $xml.CreateNode("element", "Status", "")  
$status = Get-Status -ComputerName $name  
  
if (-not $status) {  
    $statusnode.set_innertext("Unreachable")  
}  
else {  
    $statusnode.set_innertext("Complete")  
  
    # get OS info  
    $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName $name  
  
    # get local disks  
    $disks = Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName $name  
  
    # create OS node, disks node  
    $osnode = $xml.CreateNode("element", "OS", "")  
    $disksnode = $xml.CreateNode("element", "Disks", "")  
  
    # append OS attrs to node  
    $osnode.setattribute("BuildNumber", $os.buildnumber)  
    $osnode.setattribute("ServicePack", $os.servicepackmajorversion)  
  
    # append OS node to Computer node  
    $xml.computers.computer[$i].appendchild($osnode) | Out-Null  
  
    # go through the logical disks  
    foreach ($disk in $disks) {  
        # create disk node  
        $disknode = $xml.CreateNode("element", "Disk", "")  
  
        #create deviceid and freespace attrs  
        $disknode.setattribute("DeviceID", $disk.deviceid)  
        $freespace = "$($disk.freespace / 1MB -as [int])MB"  
        $disknode.setattribute("FreeSpace", $freespace)  
  
        # append Disk node to Disks node  
        $disksnode.appendchild($disknode) | Out-Null  
    }  
  
    # append Disks node to Computer node  
    $xml.computers.computer[$i].appendchild($disksnode) | Out-Null  
}  
  
# append status node to Computer node  
$xml.computers.computer[$i].appendchild($statusnode) | Out-Null  
}  
  
# output XML  
Remove-Item -Path "C:\test\inventory-out.xml" -ErrorAction SilentlyContinue  
$xml.save("C:\test\inventory-out.xml")  
  
Write-Output "...Inventory Complete."
```

## Working with XML documents

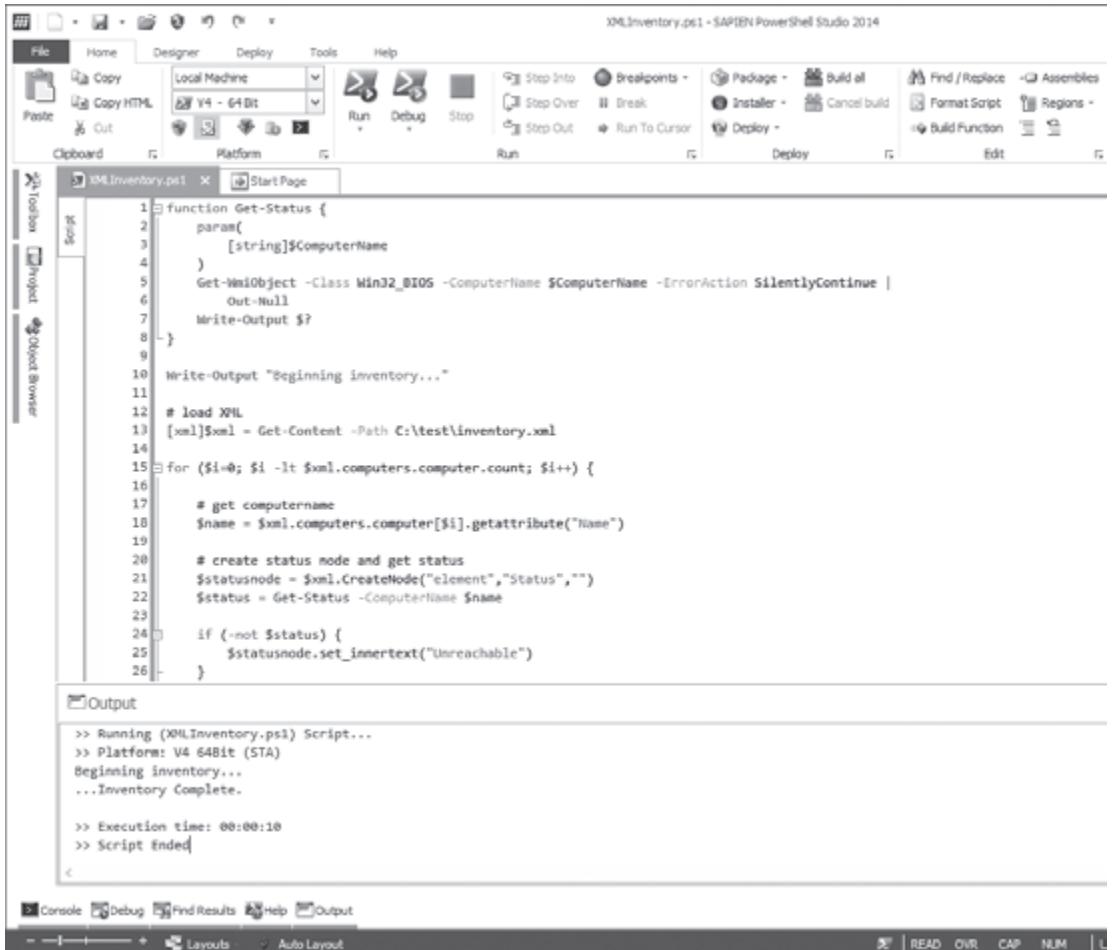


Figure 15-16

## Turning objects into XML

PowerShell also supports the **ConvertTo-XML** cmdlet. By piping objects to this cmdlet, you can have PowerShell automatically construct an XML document based on those objects' properties. You can then manipulate the resulting XML document as described in this chapter.

```
[xml]$xml = Get-Process | ConvertTo-XML
```

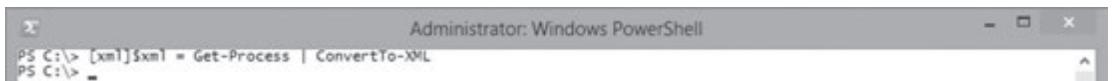
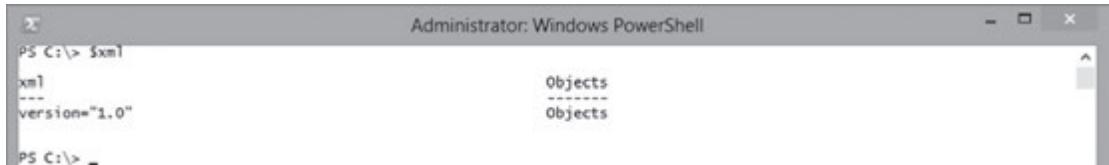


Figure 15-17

### Windows PowerShell: TFM

This cmdlet is a useful way of taking complex data represented by objects and turning it into a form that can be more easily manipulated and used for other purposes. Be aware that this XML file is different than what you get when you use the **ExportTo-Clixml** cmdlet. The **ConvertTo-XML** cmdlet is creating traditional XML.

```
$xml
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$xml is run, resulting in the following output:

```
PS C:\> $xml
xml
---
version="1.0"
PS C:\>
```

The output shows the XML structure with the root element "xml" and its attribute "version". The XML is displayed in a hierarchical tree view with the root node "xml" expanded, showing its child node "Objects". The "Objects" node has two child nodes, both also labeled "Objects".

Figure 15-18

```
$xml.Objects.Object[0]
```



A screenshot of the Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$xml.Objects.Object[0] is run, resulting in the following output:

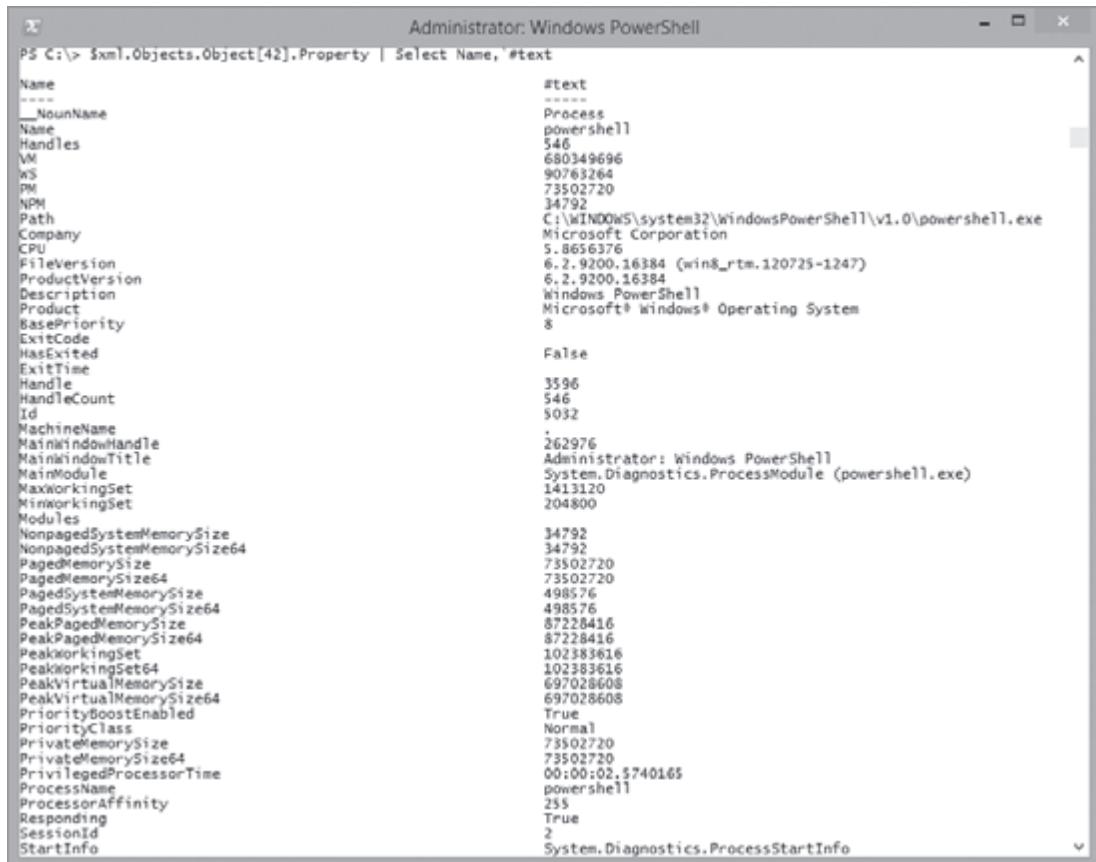
```
PS C:\> $xml.Objects.Object[0]
Type
-----
System.Diagnostics.Process
PS C:\>
```

The output shows the properties of the first object in the "Objects" array. The "Type" property is set to "System.Diagnostics.Process". The "Property" section lists several properties: \_\_NounName, Name, Handles, VM..., and others.

Figure 15-19

```
$xml.Objects.Object[0].Property | Select Name,`#text
```

## Working with XML documents



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> \$xml.Objects[42].Property | Select Name, #text is run, displaying a list of properties and their values for a process object. The properties listed include Name, Handles, VM, WS, PM, NPM, Path, Company, CPU, FileVersion, ProductVersion, Description, Product, BasePriority, ExitCode, HasExited, ExitTime, Handle, HandleCount, Id, MachineName, MainWindowHandle, MainWindowTitle, MainModule, MaxWorkingSet, MinWorkingSet, Modules, NonpagedSystemMemorySize, NonpagedSystemMemorySize64, PagedMemorySize, PagedMemorySize64, PagedSystemMemorySize, PagedSystemMemorySize64, PeakPagedMemorySize, PeakPagedMemorySize64, PeakWorkingSet, PeakWorkingSet64, PeakVirtualMemorySize, PeakVirtualMemorySize64, PriorityBoostEnabled, PriorityClass, PrivateMemorySize, PrivateMemorySize64, PrivilegedProcessorTime, ProcessName, ProcessorAffinity, Responding, SessionId, and StartInfo. The values for most properties are shown as text, while some like ProcessName and StartInfo are shown as objects.

Name	#text
Handles	546
VM	680349696
WS	90763264
PM	73502720
NPM	34792
Path	C:\WINDOWS\system32\WindowsPowerShell\v1.0\powershell.exe
Company	Microsoft Corporation
CPU	5.8656376
FileVersion	6.2.9200.16384 (win8_rtm.120725-1247)
ProductVersion	6.2.9200.16384
Description	Windows PowerShell
Product	Microsoft® Windows® Operating System
BasePriority	8
ExitCode	
HasExited	False
ExitTime	
Handle	3596
HandleCount	546
Id	5032
MachineName	:
MainWindowHandle	262976
MainWindowTitle	Administrator: Windows PowerShell
MainModule	System.Diagnostics.ProcessModule (powershell.exe)
MaxWorkingSet	1413120
MinWorkingSet	204800
Modules	
NonpagedSystemMemorySize	34792
NonpagedSystemMemorySize64	34792
PagedMemorySize	73502720
PagedMemorySize64	73502720
PagedSystemMemorySize	498576
PagedSystemMemorySize64	498576
PeakPagedMemorySize	87228416
PeakPagedMemorySize64	87228416
PeakWorkingSet	102383616
PeakWorkingSet64	102383616
PeakVirtualMemorySize	697028608
PeakVirtualMemorySize64	697028608
PriorityBoostEnabled	True
PriorityClass	Normal
PrivateMemorySize	73502720
PrivateMemorySize64	73502720
PrivilegedProcessorTime	00:00:02.5740165
ProcessName	powershell
ProcessorAffinity	255
Responding	True
SessionId	2
StartInfo	System.Diagnostics.ProcessStartInfo

Figure 15-20

As you see, we can work with this object like any other XML object.



## In Depth 16

# Windows PowerShell workflow

### Windows PowerShell workflow

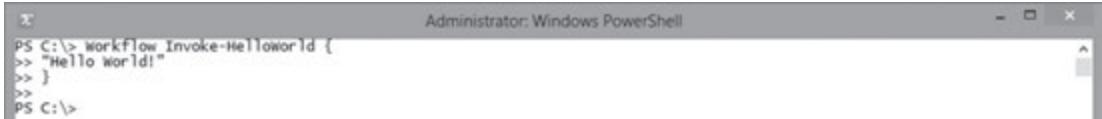
In a perfect world where computers never restart and network interruptions don't occur, PowerShell scripts, along with functions, when combined with PowerShell remoting, would suffice for automating administrative tasks. Unfortunately, we live in a less than perfect world. As IT professionals, we're sometimes required to automate administrative tasks that require long-running processes to complete successfully across multiple computers in specific sequences or in parallel, even when these sorts of issues occur.

Workflows, which were first introduced in PowerShell version 3, are a new type of PowerShell command that is designed to meet this need. A workflow is a list of PowerShell statements that are designed to perform tasks, called activities. The syntax of workflows is similar to functions. There are a number of differences between workflows and functions, but since they're similar, it allows you to leverage your existing skill set and learn the differences, instead of having to learn a completely new set of skills.

The **Workflow** keyword is used to define a workflow. Following the workflow keyword is the name that will be used for the workflow which should use the cmdlet-style naming scheme, starting with an approved verb, followed by a dash, and then finally a singular noun. The name is followed by a script block which is an opening and closing curly brace ( {} ). The commands that the workflow will execute are contained within these curly braces.

Let's go ahead and create a simple workflow so you can see the syntax. We'll use the verb **Invoke** followed by the noun **HelloWorld**.

```
Workflow Invoke-HelloWorld {
    "Hello World!"
}
```



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command entered is 'Workflow Invoke-HelloWorld { "Hello World!" }'. The output shows the definition of the workflow, which consists of a single action block that outputs 'Hello World!'.

Figure 16-1

The commands shown in Figure 16-1 defined the **Invoke-HelloWorld** workflow in our current PowerShell session, but they don't actually run the workflow.

The **Get-Command** cmdlet can be used to view workflows that have been created.

```
Get-Command - CommandType Workflow
```



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command entered is 'Get-Command - CommandType Workflow'. The output is a table showing two workflows: 'Invoke-AsWorkflow' and 'Invoke-HelloWorld', both of which are part of the module 'PSWorkflowUtility'.

CommandType	Name	ModuleName
workflow	Invoke-AsWorkflow	PSWorkflowUtility
workflow	Invoke-HelloWorld	

Figure 16-2

Keep in mind that our **Invoke-HelloWorld** workflow is only defined in our current PowerShell console session and once we close out of that PowerShell console session, the workflow will disappear. How can our workflows be made available to us each time we open PowerShell?

The solution is the same as for functions: add them to a module. As long as the module we created and added our workflow to exists in the **\$env:PSModulePath** and the **\$PSModuleAutoloadingPreference** hasn't been changed, the workflow will automatically be available each time we open PowerShell. The workflow could also be added to a script and dot-sourced for testing purposes, which is also just like what we would do with functions. Like we said before, we're leveraging as much of our existing skill set as possible here.

When you create a workflow, it's compiled into Extensible Application Markup Language (XAML). You can use the **Get-Command** cmdlet to view the XAML definition that's created for a workflow.

```
Get-Command -Name Invoke-HelloWorld | Select-Object -ExpandProperty XamlDefinition
```

## Windows PowerShell workflow



```
Administrator: Windows PowerShell
PS C:\> Get-Command -Name Invoke-Helloworld | Select-Object -ExpandProperty XamlDefinition
<Activity
  x:Class="Microsoft.PowerShell.DynamicActivities.Activity_2089052698"
  xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
  xmlns:sd="clr-namespace:System.Activities.Debugger;assembly=System.Activities"
  xmlns:local="clr-namespace:Microsoft.PowerShell.DynamicActivities"
  xmlns:mva="clr-namespace:Microsoft.VisualBasic.Activities;assembly=System.Activities"
  mva:VisualBasic.Settings="Assembly references and imported namespaces serialized as XML namespaces"
  xmlns:x="http://schemas.microsoft.com/winx/2006/xaml"
  xmlns:ns0="clr-namespace:System;assembly=mscorlib"
  xmlns:ns1="clr-namespace:Microsoft.PowerShell.Utility.Activities;assembly=Microsoft.PowerShell.Utility.Activities"
  xmlns:ns2="clr-namespace:Microsoft.PowerShell.Activities;assembly=Microsoft.PowerShell.Activities"
  xmlns:ns3="clr-namespace:System.Activities;assembly=System.Activities"
  xmlns:ns4="clr-namespace:System.Management.Automation;assembly=System.Management.Automation"
>
<Sequence>
  <ns2:SetPSScriptWorkflowData>
    <ns2:SetPSScriptWorkflowData.OtherVariableName>Position</ns2:SetPSScriptWorkflowData.OtherVariableName>
    <ns2:SetPSScriptWorkflowData.Value>
      <ns3:InArgument x:TypeArguments="ns0:Object">
        <ns2:PowerShellValue x:TypeArguments="ns0:Object" Expression="2:1:Invoke-Helloworld" />
      </ns3:InArgument>
    </ns2:SetPSScriptWorkflowData.Value>
  </ns2:SetPSScriptWorkflowData>
  <ns1:WriteOutput>
    <ns1:WriteOutput.NoEnumerate>[System.Management.Automation.SwitchParameter.Present]</ns1:WriteOutput.NoEnum
erate>
    <ns1:WriteOutput.InputObject>
      <InArgument x:TypeArguments="ns4:PSObject[]">
        <ns2:PowerShellValue x:TypeArguments="ns4:PSObject[]" Expression="Hello World!" />
      </InArgument>
    </ns1:WriteOutput.InputObject>
  </ns1:WriteOutput>
  <Sequence.Variables>
    <Variable Name="WorkflowCommandName" x:TypeArguments="ns0:String" Default = "Invoke-Helloworld" />
  </Sequence.Variables>
</Sequence>
</Activity>
PS C:\>
```

Figure 16-3

To actually execute the workflow, we need to run or call it.

Invoke-Helloworld



```
Administrator: Windows PowerShell
PS C:\> Invoke-Helloworld
Hello World!
PS C:\>
```

Figure 16-4

When you run a workflow in PowerShell, it's processed by Windows Workflow Foundation (WWF) which translates the PowerShell commands into WWF Activities. Not every PowerShell cmdlet has an equivalent WWF activity, which means that not every cmdlet can be used in a workflow directly. There is a way, however, to use these cmdlets in a workflow indirectly, or what we'll call "out of band," by placing them in something called an inline script block (which we'll cover later in this chapter). Don't worry too much about the background information or let it scare you away from workflows, you don't need to be a developer or know anything about XAML or WWF to write effective workflows in PowerShell.

In our opinion, workflows should be run as jobs from a server whenever possible, since they're designed for long-running processes that may need to be around longer than you want your PowerShell console tied up for. Also, servers are designed to be more resilient to issues, such as power outages, than your workstation. Using a console session on your workstation is fine for prototyping and testing workflows. Use the **-AsJob** parameter to run a workflow as a job, as shown in Figure 16-5. We also recommend naming the job, by using the **-JobName** parameter.

```
Invoke-Helloworld -AsJob -JobName HelloWorld
```

Administrator: Windows PowerShell						
Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
12	HelloWorld	PSWorkflowJob	Running	True	localhost	Invoke-Helloworld

Figure 16-5

Since workflows were developed for performing long-running, multi-machine orchestration, careful consideration should be taken when designing them, so that they can, if needed, be safely paused and resumed in-between activities without affecting their outcome. After all, being able to pause and resume a workflow is one of the main reasons for using a workflow in the first place. Any point where the workflow depends only on its own variables or a system state that can survive a restart is a good place to consider adding the necessary code so that the workflow can be suspended.

The **Checkpoint-Workflow** cmdlet is used to save a workflow's state to the file system so that it can be safely paused, and then resumed from the point where the checkpoint took place.

---

**Note:**

The Checkpoint-Workflow cmdlet is only valid in a workflow.

```
Workflow Invoke-Helloworld2 {
    "Hello World!"
    Start-Sleep -Seconds 30
    Checkpoint-Workflow
    "Welcome to Workflows!"
}
```

Administrator: Windows PowerShell	
PS C:\>	Workflow Invoke-Helloworld2 {
>>>	"Hello World!"
>>>	Start-Sleep -Seconds 30
>>>	Checkpoint-Workflow
>>>	"Welcome to Workflows!"
>>>	}
>>>	

Figure 16-6

There is also a **-PSPersist** workflow activity parameter that can be added to any individual command in the workflow to create checkpoints, just like the **Checkpoint-Workflow** command.

If you determine that a checkpoint can occur after every activity in the workflow, then the automatic variable **\$PSPersistPreference** can be set to **True** in the workflow or the workflow can be launched by using the **-PSPersist** parameter. Take into consideration the performance cost of collecting the data and writing it to the file system before using either of these options.

## Windows PowerShell workflow

Once the **Checkpoint-Workflow** command has been added to the workflow in one or more places, run the workflow as a job.

```
Invoke-Helloworld2 -AsJob -JobName HelloWorld2
```

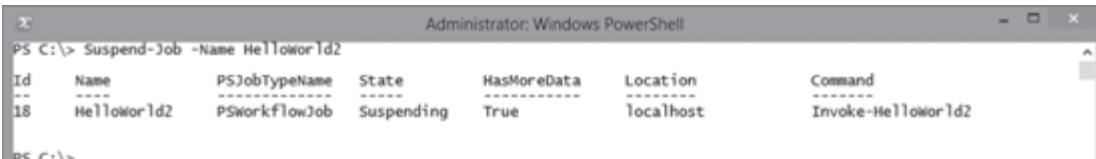


Id	Name	PSJobTypeName	State	HasMoreData	Location
2	HelloWorld2	PSWorkflowJob	Running	True	localhost

Figure 16-7

This allows you to take advantage of easily suspending the workflow, by using the **Suspend-Job** cmdlet.

```
Suspend-Job -Name HelloWorld2
```



Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
18	HelloWorld2	PSWorkflowJob	Suspending	True	localhost	Invoke-Helloworld2

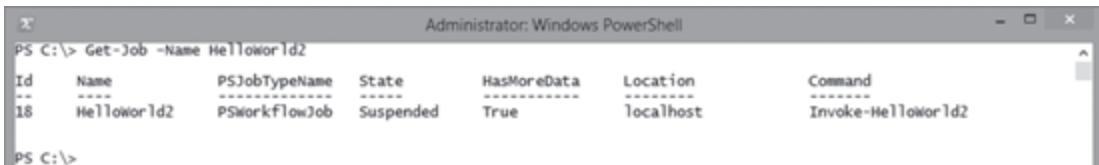
Figure 16-8

As shown in Figure 16-8, once the **Suspend-Job** cmdlet has been run, the state will be **Suspending**, and the workflow will continue to run until it reaches the next checkpoint, at which point the workflow is paused, unless the **-Force** parameter is also used. If that is the case, it will pause immediately and revert to the previous checkpoint when resumed. We don't recommend using the **-Force** parameter to forcibly suspend a workflow, unless it is absolutely necessary.

Once the next checkpoint statement is reached and the workflow's state is saved to the file system, the workflow's job will have a state of **Suspended**.

```
Get-Job -Name HelloWorld2
```

## Windows PowerShell: TFM



```
Administrator: Windows PowerShell
PS C:\> Get-Job -Name HelloWorld2
Id Name PSJobTypeName State HasMoreData Location Command
-- -- -- -- -- -- -- --
18 HelloWorld2 PSWorkflowJob Suspended True localhost Invoke-Helloworld2

PS C:\>
```

Figure 16-9

The **Suspend-Job** cmdlet is designed to only suspend workflow jobs, not standard PowerShell jobs.

To pause a workflow at a predetermined point, use the **Suspend-Workflow** cmdlet. Using this command causes a checkpoint to occur automatically before the workflow is suspended, without needing to use the **Checkpoint-Workflow** cmdlet.

```
workflow Invoke-Helloworld3 {
    "Hello World!"
    Start-Sleep -Seconds 30
    Suspend-Workflow
    "Welcome to Workflows!"
}
```

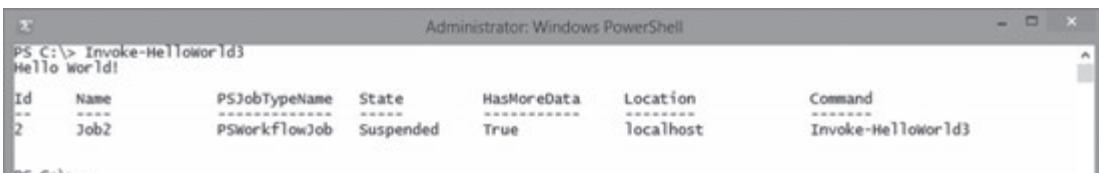


```
Administrator: Windows PowerShell
PS C:\> workflow Invoke-Helloworld3 {
>>> "Hello World!"
>>> Start-Sleep -Seconds 30
>>> Suspend-Workflow
>>> "Welcome to Workflows!"
>>>
>>>
PS C:\>
```

Figure 16-10

Even if the workflow was not run as a job, a job will be created that has a state of **Suspended**.

Invoke-Helloworld3



```
Administrator: Windows PowerShell
PS C:\> Invoke-Helloworld3
Hello World!
Id Name PSJobTypeName State HasMoreData Location Command
-- -- -- -- -- -- -- --
2 Job2 PSWorkflowJob Suspended True localhost Invoke-Helloworld3

PS C:\>
```

Figure 16-11

## Windows PowerShell workflow

One additional advantage to using the **Checkpoint-Workflow** command is if the workflow is interrupted for any reason, such as the machine restarting or the PowerShell console being closed out, a job in a suspended state will be ready to be resumed from the previous checkpoint so that the workflow doesn't have to be completely restarted.

To resume or continue a workflow from a suspended state, use the **Resume-Job** cmdlet.

```
Resume-Job -Name Job2
```

Administrator: Windows PowerShell						
Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
2	Job2	PSWorkflowJob	Suspended	True	localhost	Invoke-Helloworld3

Figure 16-12

Notice that in Figure 16-12, even though we ran the **Resume-Job** cmdlet, the state still shows **Suspended**. The job was resumed but by default the job's status is returned immediately. Use the **-Wait** parameter to have the job status update wait to return its status until the **Resume-Job** command takes effect.

Administrator: Windows PowerShell						
Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
8	Job2	PSWorkflowJob	Running	True	localhost	Invoke-Helloworld3

Figure 16-13

When resuming a workflow, it's resumed from the most recent checkpoint. There's no way to resume a workflow from a checkpoint other than the most recent one, even if multiple checkpoints exists.

Keep in mind, as with **Suspend-Job**, the **Resume-Job** cmdlet is specific to workflow jobs and can't be used with standard PowerShell jobs.

As with any other job that has completed in PowerShell, the results can be retrieved by using the **Receive-Job** cmdlet. Use the **-Keep** parameter, unless you wish to clear the results so they are no longer retrievable.

Administrator: Windows PowerShell						
PS C:\> Receive-Job -Name Job2	HELLo World!	Welcome to Workflows!	PS C:\>			

Figure 16-14

As we briefly discussed earlier, there are numerous cmdlets that are not supported within a workflow, but by using the **InlineScript** keyword, you have the ability to use the entire PowerShell scripting language. The commands inside of each **InlineScript** block execute as a single workflow activity in their own process, separate from the process of the workflow itself. Use the **\$Using** prefix within the **InlineScript** block to access variables from the workflow.

```
workflow test-inlinescript {
    $bits = Get-Service -Name BITS

    InlineScript {
        'Start without $Using'
        $bits | Format-Table -AutoSize
        'End without $Using'
    }

    InlineScript {
        "Start with $Using"
        $Using:bits | Format-Table -AutoSize
        'End with $Using'
    }
}
```

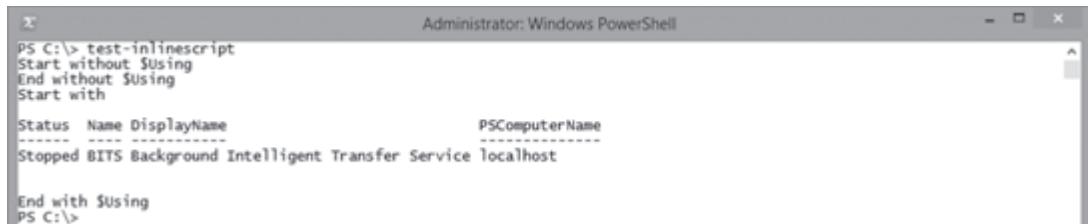
```
PS C:\> workflow test-inlinescript {
>>>     $bits = Get-Service -Name BITS
>>>
>>>     InlineScript {
>>>         'Start without $Using'
>>>         $bits | Format-Table -AutoSize
>>>         'End without $Using'
>>>     }
>>>
>>>     InlineScript {
>>>         "Start with $Using"
>>>         $Using:bits | Format-Table -AutoSize
>>>         'End with $Using'
>>>     }
>>> }
```

Figure 16-15

There are two things that we want you to notice in Figure 16-15. First, we've used the **Format-Table** cmdlet, which is not allowed in a workflow, except when used in the **InlineScript** block. Second, we've also shown that accessing a variable from the workflow inside the **InlineScript** block requires the **\$Using** prefix.

```
test-inlinescript
```

### Windows PowerShell workflow



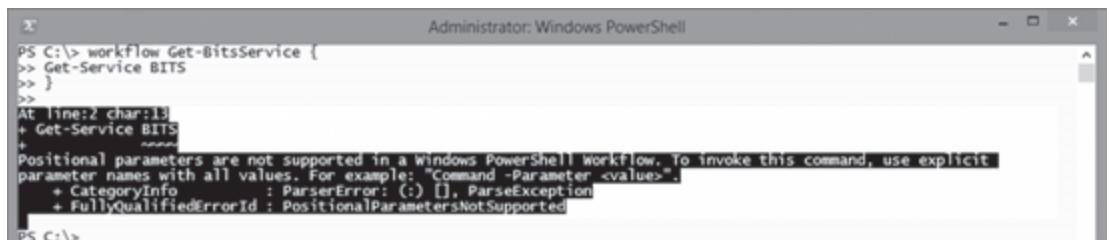
```
Administrator: Windows PowerShell
PS C:\> test-inlinescript
Start without $Using
End without $Using
Start with
Status Name DisplayName PSComputerName
----- ---- -----
Stopped BITS Background Intelligent Transfer Service localhost

End with $Using
PS C:\>
```

Figure 16-16

Positional parameters are not supported within a workflow.

```
Workflow Get-BitsService
    Get-Service BITS
}
```

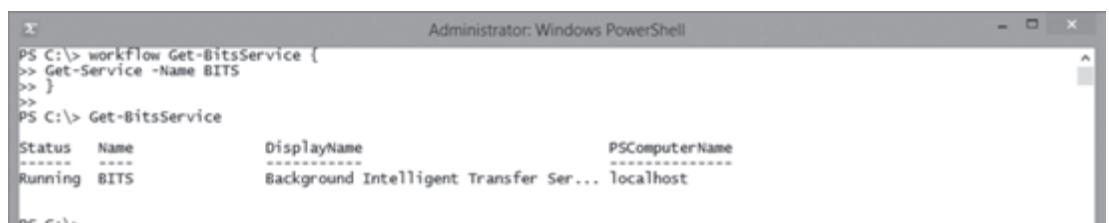


```
Administrator: Windows PowerShell
PS C:\> workflow Get-BitsService {
>>>     Get-Service BITS
>>> }
>>>
At line:2 char:13
+     Get-Service BITS
+               ~~~~~
+ Positional parameters are not supported in a Windows PowerShell Workflow. To invoke this command, use explicit
parameter names with all values. For example: "Command -Parameter <value>".
+ CategoryInfo          : ParserError: (:) [], ParseException
+ FullyQualifiedErrorId : PositionalParametersNotSupported
PS C:\>
```

Figure 16-17

You'll need to use full parameter names.

```
Workflow Get-BitsService {
    Get-Service -Name BITS
}
Get-BitsService
```



```
Administrator: Windows PowerShell
PS C:\> workflow Get-BitsService {
>>>     Get-Service -Name BITS
>>> }
>>>
PS C:\> Get-BitsService
Status   Name      DisplayName      PSComputerName
-----   --      -----      -----
Running  BITS      Background Intelligent Transfer Ser...  localhost

PS C:\>
```

Figure 16-18

The only exception to this is within an **InlineScript** block.

```
Workflow Get-BitsService2 {
    InlineScript {
        Get-Service BITS
    }
}
Get-BitsService2
```

Status	Name	DisplayName	PSComputerName
Running	BITS	Background Intelligent Transfer Ser...	localhost

Figure 16-19

Even though it's possible to use positional parameters within the **InlineScript** block, we still recommend using full parameter names, instead of positional parameters.

Let's create a test workflow.

```
workflow test-workflow {
    "Waiting 30 Seconds"
    Start-Sleep -Seconds 30
    Get-Service -Name BITS

    "Waiting 30 Seconds"
    Start-Sleep -Seconds 30
    Get-Service -Name W32Time

    "Waiting 30 Seconds"
    Start-Sleep -Seconds 30
    Get-Service -Name Spooler
}
```

Figure 16-20

## Windows PowerShell workflow

Running this workflow shows that it runs sequentially from the first command to the last command, waiting for 30 seconds between retrieving each service that we've specified.

```
test-workflow
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> test-workflow is run. The output displays the status and name of three services: BITS (Stopped), W32Time (Running), and Print Spooler (Running). Each service is listed with its status, name, display name, and PSCoputerName (localhost).

Status	Name	DisplayName	PSCoputerName
Stopped	BITS	Background Intelligent Transfer Ser...	localhost
Running	W32Time	Windows Time	localhost
Running	Spooler	Print Spooler	localhost

Figure 16-21

Now we'll see how long it takes to run this workflow.

```
Measure-Command {test-workflow}
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Measure-Command {test-workflow} is run. The output provides detailed timing information for the workflow execution, including Days, Hours, Minutes, Seconds, Milliseconds, Ticks, TotalDays, TotalHours, TotalMinutes, TotalSeconds, and TotalMilliseconds.

Days	: 0
Hours	: 0
Minutes	: 1
Seconds	: 30
Milliseconds	: 315
Ticks	: 903153568
TotalDays	: 0.00104531662962963
TotalHours	: 0.0250875991111111
TotalMinutes	: 1.505255946666667
TotalSeconds	: 90.3153568
TotalMilliseconds	: 90315.3568

Figure 16-22

It took just over 90 seconds for the workflow to complete. That makes sense since we waited 30 seconds, three different times. None of these commands depend on one another, so how can we speed up this process? What if we were able to run these commands in parallel, instead of sequentially?

The **Parallel** keyword is used to have commands in the workflow execute at the same time.

```
workflow test-parallel {
    parallel {
        "Waiting 30 Seconds"
        Start-Sleep -Seconds 30
        Get-Service -Name BITS
```

```

"Waiting 30 Seconds"
Start-Sleep -Seconds 30
Get-Service -Name W32Time

"Waiting 30 Seconds"
Start-Sleep -Seconds 30
Get-Service -Name Spooler
}

}

```

```

Administrator: Windows PowerShell

PS C:\> workflow test-parallel {
>>     parallel {
>>         "Waiting 30 Seconds"
>>         Start-Sleep -Seconds 30
>>         Get-Service -Name BITS
>>
>>         "Waiting 30 Seconds"
>>         Start-Sleep -Seconds 30
>>         Get-Service -Name W32Time
>>
>>         "Waiting 30 Seconds"
>>         Start-Sleep -Seconds 30
>>         Get-Service -Name Spooler
>>     }
>> }
>> }

PS C:\>

```

Figure 16-23

That command completed in approximately 30 seconds, which also makes sense because the three sleep states ran in parallel.

```
Measure-Command { test-parallel }
```

```

Administrator: Windows PowerShell

PS C:\> Measure-Command {test-parallel}

Days          : 0
Hours         : 0
Minutes       : 0
Seconds       : 30
Milliseconds  : 291
Ticks         : 302917397
TotalDays     : 0.00035059883912037
TotalHours    : 0.00841437213888889
TotalMinutes  : 0.504862328333333
TotalSeconds  : 30.2917397
TotalMilliseconds : 30291.7397

PS C:\>

```

Figure 16-24

The problem, though, is all of the commands ran in parallel, so the **Get-Service** statements finished almost immediately and we were waiting on each of the **Sleep** statements to finish, which isn't what we wanted. What if there was a way to group commands together and run each group in parallel?

It just so happens that there's a sequence keyword that does exactly what we need. It's used in conjunction with the **Parallel** keyword to have groups of commands execute sequentially, instead of in parallel within a parallel block. The commands in the sequence block will execute sequentially, but the sequence block itself will execute in parallel with the other commands in the workflow.

```
workflow test-sequence {
    parallel {
        sequence {
            "Waiting 30 Seconds"
            Start-Sleep -Seconds 30
            Get-Service -Name BITS
        }

        sequence {
            "Waiting 30 Seconds"
            Start-Sleep -Seconds 30
            Get-Service -Name W32Time
        }

        sequence {
            "Waiting 30 Seconds"
            Start-Sleep -Seconds 30
            Get-Service -Name Spooler
        }
    }
}
```

Figure 16-25

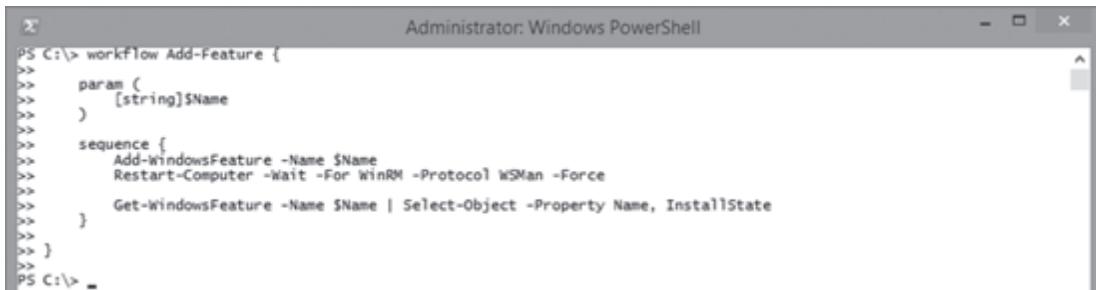
Keep in mind, there's no guarantee what order the commands will execute in when you use the **Parallel** keyword.

Workflows also include a **Foreach parallel** construct, which is a way to run commands in parallel against a collection of items, such as a collection of disks or services. The actual commands within a **Foreach parallel** block run sequentially. It runs the workflow in parallel against all of the items in the collection that is targeted by the workflow. In PowerShell version 3, the hard-coded throttle limit for the number of items to be targeted at one time is five, but in PowerShell version 4, this is a configurable setting.

There's no need to use the **Foreach parallel** construct to run a workflow in parallel against multiple computers, because when the target computers are specified by using the **-PSCoMputerName** parameter, they're always processed in parallel.

We'll now create a simple workflow which installs a Windows Feature on a server, restarts the server, waits for it to finish restarting, and then checks to make sure the feature is installed.

```
workflow Add-Feature {
    param (
        [string]$Name
    )
    sequence {
        Add-WindowsFeature -Name $Name
        Restart-Computer -Wait -For WinRM -Protocol WSMan -Force
        Get-WindowsFeature -Name $Name | Select-Object -Property Name, InstallState
    }
}
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> workflow Add-Feature { ... } is being typed into the prompt. The script itself is displayed in the window, showing the param block, sequence block, and its contents: Add-WindowsFeature, Restart-Computer, and Get-WindowsFeature cmdlets. The command is partially typed at the bottom of the window.

Figure 16-26

We'll demonstrate running this workflow against multiple computers. As we've previously mentioned, when computers are specified via the **-PSCoMputerName** parameter, the workflow will run in parallel against those computers, without needing to use a **Foreach parallel** loop.

```
Add-Feature -PSCoMputerName Server01, Server02, Server03 -Name Telnet-Client
```

## Windows PowerShell workflow

The screenshot shows an Administrator: Windows PowerShell window. The command run is PS C:\> Add-Feature -PSComputerName Server01, Server02, Server03 -Name Telnet-Client. The output shows the workflow restarting the three servers and then checking the install state.

```
Administrator: Windows PowerShell
PS C:\> Add-Feature -PSComputerName Server01, Server02, Server03 -Name Telnet-Client

- Restarting computer Server01
  Waiting for the restart to begin...
  [
Restart-Computer
  Running

  1.13: Add-Feature line:7 char:7
- Restarting computer Server03
  Waiting for the restart to begin...
  [
- Restarting computer Server02
  Waiting for the restart to begin...
  [
```

Figure 16-27

As you can see, without any user interaction other than invoking the workflow, the telnet client was installed, servers restarted, and then the install state was checked after the servers were back online.

The screenshot shows an Administrator: Windows PowerShell window. The command run is PS C:\> Add-Feature -PSComputerName Server01, Server02, Server03 -Name Telnet-Client. The output displays the success of each step, the feature result, and the target computer name, along with several warning messages about automatic updating.

Success	Restart Needed	Exit Code	Feature Result	PSComputerName
True	No	Success	{Telnet Client}	Server01
WARNING: [Server01]:Windows automatic updating is not enabled. To ensure that your newly-installed role or feature is automatically updated, turn on Windows Update.				
True	No	Success	{Telnet Client}	Server03
WARNING: [Server03]:Windows automatic updating is not enabled. To ensure that your newly-installed role or feature is automatically updated, turn on Windows Update.				
True	No	Success	{Telnet Client}	Server02
WARNING: [Server02]:Windows automatic updating is not enabled. To ensure that your newly-installed role or feature is automatically updated, turn on Windows Update.				
Name : Telnet-Client				
InstallState : Installed				
PSComputerName : Server01				
PSSourceJobInstanceId : f0c44aed-f17b-4d82-8d24-a512952260ff				
Name : Telnet-Client				
InstallState : Installed				
PSComputerName : Server03				
PSSourceJobInstanceId : 26097327-a098-40c3-b67c-4a28b7750022				
Name : Telnet-Client				
InstallState : Installed				
PSComputerName : Server02				
PSSourceJobInstanceId : 33f5424d-a564-4dd1-87ae-108c86114622				

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

Figure 16-28

If you're interesting in learning more about PowerShell Workflows, we recommend reading the About\_workflows Help topic in PowerShell.



# Index

## A

Active Directory PowerShell module, 370, 561  
 Active Directory properties, 573–574  
 Add-History cmdlet, 29–30  
 adding snap-ins, 169–170, 170*f*  
 Add-PSSnapin cmdlet, 170  
 AddToGroup.ps1, 586–587  
 Add-WindowsFeature cmdlet, 13, 680  
 ADSI queries, 560  
 advanced functions, 287–289, 288–289*f*  
 aliases, 26–29, 27*f*, 29*f*, 151, 450–451  
 AliasProperty extension, 77  
 AllSigned execution policy, 156–157, 160, 165  
 API (applications programs interface), 369  
 -Append parameter, 440  
 applications programs interface (API), 369  
 arrays

- Active Directory properties, 573–574
- associative, 190–192, 191–192*f*
- collection, 187–189, 188*f*
- working with, 189–192, 189–192*f*

ASDI (Active Directory Service Interfaces)  
 bulk-creating users, 582–584  
 creating groups, 586–587  
 creating new computer accounts, 584–585  
 deleting computer accounts, 585  
 deleting users, 581  
 fundamentals, 559–560

LDAP filters use, 588–590

Microsoft Active Directory cmdlets, 590–592  
 objects  
   moving, 587  
   retrieving, 561–567, 562–567*f*  
   searching for, 567–572, 569*f*, 571–572*f*  
   using, 561, 563  
   working with, 573–574  
 obtaining password age, 580–581  
 queries, 560  
 searching for users, 587–588  
 type adapters, 576–580, 577*f*, 579–580*f*  
   using .NET Framework DirectoryService, 575  
 ASDI type adapters, 576–580, 577*f*, 579–580*f*  
 assignment operators, 551–552*t*, 551–557, 552–554*f*, 556–557*f*  
 automatic variables, 181

## B

background and scheduled jobs  
 defined, 259–260  
 local background jobs, 260–261, 260–261*f*  
 managing jobs, 263–268, 263–268*f*  
 remote background jobs, 261–263, 261–263*f*  
 scheduled jobs  
   options, 271–273, 271–273*f*  
   registering and using, 275–278, 275–278*f*  
   triggers, 274–275, 274–275*f*

storing output from jobs, 268–270, 269–270/*f*  
 BackupEventLog() method, 223, 432–434, 437–438  
*Backup-EventLog.ps1*, 434–438  
 breakpoints, advanced debugging, 343–345, 343–345/*f*  
 Break statement, 210–211, 210–211/*f*  
 built-in Help, 44, 72, 161, 215  
*BulkCopy.ps1*, 449–450, 452  
 bulk-creating users, 582–584  
*BulkRename.ps1*, 451–452  
 Bypass execution policy, 156

**C**

*Cacls.exe* command-line utility, 465, 467, 472–473  
 case sensitivity, 107, 107/*f*  
 Catch block, 358  
*Cd* command, 24–25, 68–69, 508–509  
 certificate store, 67, 160  
*ChangeACL.ps1*, 470–471  
 Change() method, 490  
 changing location, 68–69, 68–69/*f*  
*ChDir* command, 68  
 Checkpoint-Workflow cmdlet, 670–671, 673  
 child items, 63–67  
 CIM (Common Information Model)  
     CIMSessions creation, 226–232  
     navigating and using, 215–216, 216/*f*  
     versus WIM, 214–215, 214–215/*f*  
 -CimSession parameter, 414–416  
*Clear-EventLog* cmdlet, 223–224, 434–438  
*ClearEventLog()* method, 434–438  
*Clear-Item* cmdlet, 70–71  
*Clear-Variable* cmdlet, 182–183  
*Cmd.exe*, 450–451  
 cmdlets  
     anatomy, 44–47, 44–47/*f*  
     changing locations, 63  
     control, 44–45/*f*  
     creating variables with cmdlets, 178–183, 180–182/*f*  
     debugging, 342, 342/*f*  
     exporting, 619/*f*  
     formatting data, 141, 143–146, 143–146/*f*, 251, 313, 608, 610  
     importing, 132–134, 133/*f*, 618–619, 619/*f*  
     Join-Path cmdlet, 461  
     listing child items, 63  
     piping, 88–97, 88–97/*f*, 274, 313  
     PowerShell core, 25–26, 25–26/*f*  
     snap-in cmdlets, 169–170, 170/*f*  
     syntax, 44–46, 44–46/*f*, 47–50, 48–50/*f*  
     transcript cmdlets, 32–33, 32–33/*f*  
     variables creation, 178–183  
         . *See also specific cmdlet*

collection arrays, 187–198, 188/*f*  
 collections, comparing, 134–136, 135–136/*f*  
 command completion, 31  
 command-line utilities, 2, 465, 467, 472–473  
 commands  
     command line assistance, 29–32  
         command history, 29–31, 30/*f*  
         copy and paste, 31  
         line editing, 31  
         tab completion, 31–32  
     manipulating PowerShell, 23–25, 24/*f*

. *See also cmdlets*  
 comma-separated value (CSV) files, 127–132, 128–132/*f*, 136–138, 137/*f*  
 comment-based help, 325–332, 328–332/*f*  
 Common Information Model. *See CIM*  
 common verbs, 26  
 COM objects, 2  
 Compare-Object cmdlet, 135–136, 135–136/*f*  
 comparing objects and collections, 134–136, 135–136/*f*  
 comparison operators, 117–118/*f*, 117–119  
 Compress() method, 456  
 -ComputerName parameter, 366  
 ComputerName variable, 339  
 ConfigureEventLog() method, 432  
 Connect-PSSession cmdlet, 245, 245/*f*  
 constructs, 437, 652  
 Continue statement, 210–211, 211/*f*  
 Convert-Path cmdlet, 457–458, 458/*f*  
 ConvertTo-CSV cmdlet, 136–138, 137/*f*  
 ConvertTo-HTML cmdlet, 138–141, 138–141/*f*  
 ConvertTo-XML cmdlet, 136–138, 137/*f*, 663–665, 663–665/*f*  
 Copy-Item cmdlet, 70–71, 70–71/*f*, 448–450  
*CreateNewComputer.ps1*, 584–585  
*Create-ServerReport.ps1*, 644–645  
*CreateUser.ps1*, 577–580  
 creating directories, 462  
 -Credential parameter, 450  
 cryptographic hash, 158  
 CSV (comma-separated value files), 127–132, 128–132/*f*, 136–138, 137/*f*  
 custom objects, 305–311, 310–311/*f*  
 custom views, 141, 313, 322

**D**

data  
     exporting and formatting  
         comma-separated value (CSV) files, 127–132, 128–132/*f*, 136–138, 137/*f*  
         comparing objects and collections, 134–136, 135–136/*f*  
         converting to CSV and XML, 136–138, 137/*f*  
         converting to HTML, 138–141, 138–141/*f*  
         exporting and importing XML, 132–134, 133/*f*  
         formatting lists and tables, 143–147, 143–147/*f*  
         formatting objects for reports, 141–143, 142/*f*  
         grid view data formatting, 148–150/*f*, 148–151  
         grouping information, 147–148, 147/*f*  
         Import- and Export- other delimited data, 132  
         useful reporting, 127  
     filtering, 115–116, 115–116/*f*  
     storage  
         associative arrays, 190–192, 191–192/*f*  
         collection arrays, 187–198, 188/*f*  
         creating variables with cmdlets, 179–183, 180–182/*f*  
         defining variables, 175–178, 176–177/*f*  
         forcing a data type, 183–187, 184–186/*f*  
         handling variables inside quotes, 192–194/*f*, 192–195  
         scripting, 178–179, 178–179/*f*  
         variables with multiple objects, 187–192, 188–192/*f*  
         working with individual elements in an array, 189–192, 189–192/*f*  
 data types, forcing, 183–187, 184–186/*f*  
 debugging and error handling

- cmdlets, 342, 342*f*  
 displaying errors with Write-Warning and Write-Error, 360–362, 360–362*f*  
 \$ErrorActionPreference variable, 345  
 generic error handling, 346  
`Get-SystemInventory.ps1`, 352–354  
 handling error output, 355  
 inline debugging with Write-Debug and Write-Verbose, 350–354, 350–354*f*  
 purpose, 341  
 setting breakpoints, 343–345, 343–345*f*  
 storing errors in a log for later, 362–367, 362–367*f*  
 using -ErrorAction and -ErrorVariable parameters, 346–350, 346–350*f*  
 using Try-Catch-Finally block, 355–360, 355–360*f*  
-Debug parameter, 338, 352  
`Decrypt()` method, 455  
 default value, 241, 290, 293, 360, 524, 585  
 defining variables, 175–178, 176–177*f*  
 deleting computer accounts, 585  
 deleting directories, 463  
 deleting files, 4  
 deleting users, 581  
-Delimiter parameter, 132  
Desired State Configuration (DSC)  
 configuration MOF file, 409–411  
 configuration reusability, 409  
 deployment, 412–418, 412–418*f*  
 MOF file use, 403–404  
`PSDesiredStateConfigurations` cmdlets, 403  
 resources, 404–408, 404–408*f*  
-Detailed parameter switches, 41–42, 41–42*f*  
 digitally signing scripts, 159–160  
Dir command, 24–25, 63, 155, 463  
directories  
 creating, 462  
 deleting, 463, 463*f*  
 listing, 463  
 services  
 ADSI queries, 560  
 bulk-creating users, 582  
 deleting users, 581  
 Microsoft Active Directory cmdlets, 590–592  
 moving objects, 587  
 password ages, 580–581  
 WinNT://Provider, 560–564  
Disable-PSBreakpoint cmdlet, 345  
Disable-ScheduledJob cmdlet, 278  
Disconnect-PSSession cmdlet, 245, 245*f*  
documentation  
 Help system, 40, 325–332  
 Inline, 335–339  
 MSDN, 490–491, 567  
 WMI, 215, 607  
-DontShow parameter, 293–294  
dot sourcing scopes, 634–635  
Do/Until statement, 208–210, 209*f*  
Do/While statement, 208, 208*f*  
Dscls.exe command-line utility, 465  
DSC. *See* Desired State Configuration
- E**  
\$EnableErrorDetailedLog parameter, 364  
\$EnableErrorSummaryLog parameter, 364  
EnablePSBreakpoint cmdlet, 345  
Enable-PSRemoting cmdlet, 236–237, 236–237*f*  
enabling PowerShell remoting, 236–237  
Encrypt() method, 455  
Enter-PSSession cmdlet, 238, 238*f*, 242–243, 242–243*f*  
-EntryType parameter, 424  
enumerating keys, 518  
EnumKeys() method, 518  
EnumValues() method, 518–519  
environment variables, 56, 69  
-ErrorAction parameter, 346–350, 346–350*f*, 517, 658  
\$ErrorActionPreference variable, 345, 360  
ErrorDetailLog variable, 362  
error handling. *See* debugging and error handling  
ErrorSummaryLog variable, 362  
-ErrorVariable parameter, 346–350, 346–350*f*  
events  
 event logs  
 backup event logs, 432–433  
 clearing event logs, 434–438  
 configuring event logs, 432  
 location, 433–434  
 managing, 421–426, 421–426*f*  
 remote event logs, 430–431  
 Windows 7 and later, 427–430, 427–430*f*  
WMI  
 actions, 610–613, 610–613*f*  
 practical uses, 607–610, 609*f*  
 querying for specific events, 613–614  
 removing, 613  
 unregistering, 613  
event subscribers, 611  
-Exclude parameter, 65, 488  
execution policies, 156–157, 160, 165  
-ExpandProperty parameter, 81, 81*f*, 388  
Export-Alias cmdlet, 29  
Export-CliXML cmdlet, 132–134, 133*f*, 617–618, 617–618*f*  
Export-Csv cmdlet, 127–128, 128*f*, 130, 130*f*, 132  
exporting and importing XML, 132–134, 133*f*  
exporting XML files, 617–620*f*, 617–621  
Export- other delimited data, 132  
Extensible Application Markup Language (XAML), 668–669, 669*f*
- F**  
file attributes and properties, 452–454, 454*f*, 455*f*  
files and folders management  
 Convert-Path cmdlet, 457–458, 458*f*  
 copying files, 448–450  
 creating directories, 462  
 creating text files, 439–441, 439–441*f*  
 deleting directories, 463, 463*f*  
 deleting files, 4, 450–451  
 file attributes and properties, 452–454, 454*f*  
 file compression, 456  
Join-Path cmdlet, 437, 461  
LastAccessTime property, 455–456  
listing directories, 463  
parsing INI files, 446–448

parsing ISS log files, 443–445  
 parsing text files, 443  
 provider alert, 450  
 reading text files, 441–442  
 renaming files, 451–452  
*Resolve-Path* cmdlet, 460–462, 461*f*  
 setting attributes, 454–455, 455*f*  
*Split-Path* cmdlet, 458–460, 459–460*f*  
*Test-Path* cmdlet, 457, 457*f*  
 working with paths, 457–462  
*-Filter* parameter, 65, 231, 602, 659  
 filters, 429, 588–590  
*FindUserDN.ps1*, 589–590  
*Find-UserFunction.ps1*, 567–568  
*Find-UserRevised.ps1*, 572  
*-Force* parameter, 486–487  
 forcing a data type, 183–187, 184–186*f*  
*ForEach* alias, 312  
*ForEach* loop, 204–206, 205–206*f*, 312, 472–473  
*ForEach-Object* cmdlet, 81–85, 82–85*f*, 204, 213, 606  
*ForEach-Object* loop, 312  
*Foreach* parallel construct, 679–680  
*Foreach* statement, 204–206, 205–206*f*, 426  
*For* loop, 206–207, 207*f*  
*Format-Custom* cmdlet, 141  
*Format-List* cmdlet, 141, 143–144, 143–144*f*, 251  
*Format-Table* cmdlet, 141–146, 144–146*f*, 313, 608, 610, 674  
 formatting  
   lists and tables, 143–147, 143–147*f*  
   objects for reports, 141–143, 142*f*  
*Format-Wide* cmdlet, 141, 146–147, 147*f*  
 For statement, 206–207, 207*f*  
 functions  
   advanced, 287–289, 288–289*f*  
   scripting, 287–289, 288–289*f*

**G**

*Get-Acl* cmdlet, 466–470*f*, 469–470  
*Get-Alias* cmdlet, 26–27, 27*f*  
*Get-ChildItem* cmdlet, 63–67, 64–66*f*, 111, 111*f*, 453, 463, 473, 515  
*Get-CimClass* cmdlet, 220, 225  
*Get-CIMInstance* cmdlet, 214–215, 214–215*f*, 220–222, 221–222*f*, 226  
*Get-CimSessions* cmdlet, 226–227  
*Get-Command* cmdlet, 247, 247*f*, 285  
*Get-Connect* cmdlet, 441  
*Get-Content* cmdlet, 189–190, 190*f*, 441, 446, 608  
*Get-Credential* cmdlet, 601  
*Get-Culture* cmdlet, 641  
*Get-Date* cmdlet, 642  
*Get-DscConfiguration* cmdlet, 414–415  
*Get-Event* cmdlet, 611–613, 611–613*f*  
*Get-EventLog* cmdlet, 421–426*f*, 421–427, 431  
*Get-ExecutionPolicy* cmdlet, 165  
*Get-Help\*CIM\** cmdlet, 594  
*Get-Help* cmdlet, 39–41, 39–41*f*, 44–45, 44–45*f*, 331  
*Get-Help* parameter, 325  
*Get-History* cmdlet, 30–31, 30*f*  
*Get-Item* cmdlet, 70  
*Get-ItemProperty* cmdlet, 509–510, 515–517  
*Get-Job* cmdlet, 263–264, 263–264*f*, 266, 266*f*, 277, 277*f*

*Get-JobTrigger* cmdlet, 274  
*Get-LocalUser* function, 355  
*Get-LocalUser.ps1*, 283–285, 283–285*f*  
*Get-Member* cmdlet, 77–78, 77*f*, 104, 176, 178, 565, 567, 603  
*Get-Module* cmdlet, 171, 172*f*  
*Get-Owner* cmdlet, 473–475, 474–475*f*  
*GetOwner()* method, 500–501  
*Get-PrincipalFilter.ps1*, 477  
*Get-Process* cmdlet, 103–104, 103–104*f*, 128–129, 128–129*f*, 141–143, 142*f*, 493–495, 493–495*f*, 500, 502–503  
*Get-PSDrive* cmdlet, 55, 56*f*, 57*f*, 62, 507  
*Get-PSProvider* cmdlet, 57–58, 58*f*  
*Get-PS.ps1*, 504–505  
*Get-PSSnapin* cmdlet, 169–170, 170*f*  
*Get-RegistryPath.ps1*, 519–521  
*Get-Required.ps1*, 482  
*Get-ScheduledJob* cmdlet, 274  
*Get-ScheduledJobOption* cmdlet, 272, 273*f*  
*Get-Service* cmdlet, 26, 92–94, 92–94*f*, 104–105, 104–105*f*, 146–147, 177, 480, 488  
*GetStringValue()* method, 519  
*Get-SystemInventory* function, 319, 322, 350–354  
*Get-SystemInventory.ps1*, 352  
*Get-Variable* cmdlet, 180–181, 181*f*  
*Get-Verb* cmdlet, 26, 26*f*  
*Get-WindowsFeature* cmdlet, 242  
*Get-WinEvent* cmdlet, 427–428, 430  
*Get-WmiObject* cmdlet, 144, 214–215, 214–215*f*, 217–218, 218*f*, 226, 433–434, 437, 488–490, 488–490*f*, 596–597, 607, 610  
 graphical user interfaces, 2  
 grid view data formatting, 148–150*f*, 148–151  
*-GroupBy* parameter, 148  
 grouping information, 147–148, 147*f*  
*GroupObject* cmdlet, 148

**H**

handling variables inside quotes, 192–194*f*, 192–195  
 help  
   adding tools, 325–339  
   cmdlet anatomy, 44–47, 44–47*f*  
   commands, 39–41, 39–41*f*  
   comment-based help, 325–332, 328–332*f*  
   inline documentation with *Write-* cmdlets  
     *Write-debug* statement, 337–339, 337–339*f*  
     *Write-Verbose* statement, 335–336, 335–336*f*, 350–354, 350–354*f*  
   parameter, 333–335, 333–335*f*  
   parameter sets, 47–48, 48*f*  
   parameter switches, 41–42, 41–42*f*  
   parameter syntax, 48–50, 48–50*f*  
   positional parameters, 46, 46*f*  
   PrimalScript 2014, 50–51, 50–51*f*  
   shortening parameter names, 47, 47*f*  
   topic oriented help files, 42–44, 43–44*f*  
   updating, 38–44  
 hierarchical storage, 55–57, 507  
 \$host variable, 639–641, 640–641*f*, 649–650  
 HTML  
   converting objects to, 138–141, 138–141*f*  
   exporting and importing, 132–134, 133*f*

**I**

(If) statement, 197–202, 198–202*f.* 203  
 implicit remoting, 246–248  
 Import-Alias cmdlet, 29  
 Import- and Export- other delimited data, 132  
 Import-CliXML cmdlet, 618–619, 619*f.*  
 Import-CSV cmdlet, 128–129, 128–129*f.* 131–132, 131–132*f.*  
 importing modules, 170–172, 171–172*f.*  
 importing XML files, 618–619, 619*f.*  
 Import-Module cmdlet, 171, 172*f.*  
 Import-PSSession cmdlet, 247, 247*f.*  
 Import-Users.ps1, 582–584  
 -Include parameter, 65  
 information selection, 78–81, 78–81*f.*  
 INI files, parsing, 446–448  
 InlineScript keyword, 674, 676  
 installation  
     PowerShellStudio, 18  
     PrimalScript, 13  
     WMI, 8–9  
 Integrated Scripting Environment, 10, 12–13, 12*f.* 153  
 Internet safe scripts, 160–161, 161*f.*  
 Invoke-Command cmdlet, 239–241*f.* 239–243, 242–243*f.* 246,  
     246*f.* 252–256, 253–256*f.* 278  
 Invoke-Item command, 70  
 Invoke-WebRequest cmdlet, 653  
 Invoke-WMIMethod cmdlet, 491, 505, 607  
 IP addresses, 533–534, 533–534*f.*  
 ISE (Integrated Scripting Environment), 10, 12–13, 12*f.* 153  
 item manipulation cmdlets, 70–72, 70–72*f.*

**J**

Job cmdlets  
     Get, 263–264, 263–264*f.* 266, 266*f.* 277, 277*f.*  
     Receive, 263–264*f.* 267*f.* 277, 673  
     Remove, 267, 267*f.*  
     Resume, 673  
     Start, 260–261, 260–261*f.*  
     Suspend, 671–672  
 Join-Path cmdlet, 437, 461

**L**

LastAccessTime property, 455–456  
 LDAP filters, 588–590  
 least privilege principle, 160–161  
 -Like operator, 120  
 listing, directories, 463  
 listing services, 479–482, 480–481*f.*  
 lists, 143–144  
 List-WMIProperties.ps1, 597  
 List-WMIVValues.ps1, 599  
 local background jobs, 260–261, 260–261*f.*  
 logical operators, 120–122, 121*f.* 453  
 logic errors, 357  
 logon account, change service, 490–491, 491*f.* 492  
 longest running processes, 498–499  
 loops, building, 15

**M**

Makecert.exe, 159–160  
 Managed Object Format (MOF) file, 403–404, 409–411,  
     410–411*f.*  
 Matches() method, 445  
 -Match operator, 120  
 Md command, 24  
 Measure-Command cmdlet, 677  
 Measure-Object cmdlet, 110–112  
 memory usage, processes, 498  
 Microsoft Active Directory cmdlets, 590–592  
 Microsoft Common Engineering Criteria, 2  
 Microsoft .NET Framework Directory Service. *See .NET*  
     Framework  
 MMC (Microsoft Management Console), 167–168, 168*f.*  
 modules  
     creating, 169  
     importing, 170–172, 171–172*f.* 232  
     scripts versus, 294  
     snap-ins and, 169  
     turning functions into, 294–295  
     workflow, 668  
 MOF (Managed Object Format) file, 403–404, 409–411,  
     410–411*f.*  
 Move-Item cmdlet, 70–71  
 \$MyVar variable, 175–178, 177–178*f.*

**N**

-Name parameter, 408  
 namespaces, 213, 560  
 navigation commands, 24–25, 63, 68–69, 155, 508–509  
 NegativeMatchingTest.ps1, 549  
 nested prompts, 637–638, 647–648  
 .NET Framework  
     adding remote registries, 523  
     directory object searching, 560, 565, 567  
     forcing a data type, 183  
     grouping categories, 537  
     language, 169  
     managing directory services, 575  
     moving objects, 587  
     registry management, 523–524  
     regular expressions, 526  
     searching for users, 587–588  
     Windows forms, 369  
 .NET Framework Software Development Kit (SDK), 159  
 New-Alias cmdlet, 630, 635  
 New-CimSessionsOption, 227  
 New-Item cmdlet, 71–72, 72*f.* 440–441, 462, 462*f.* 512–514  
 New-ItemProperty cmdlet, 513–514  
 New-JobTrigger cmdlet, 274  
 New-ModuleManifest cmdlet, 295–298  
 New-Object cmdlet, 307  
 New-PSDrive cmdlet, 55, 62, 630  
 New-PSSession cmdlet, 242, 244  
 New-ScheduledJobOption cmdlet, 272–273, 272–273*f.*  
 New-ScheduledTaskOption cmdlet, 271  
 New-SMBMapping cmdlet, 62–63, 63*f.*  
 New-Variable cmdlet, 181, 630  
 \*nix operating systems, 1, 24, 68, 75, 153  
 -NoClobber parameter, 440  
 NonTerminatingError.ps1, 346–347

objects  
 ASDI  
   moving, 587  
   retrieving, 561–567, 562–567*f*  
   searching for, 567–572, 569*f*, 571–572*f*  
   using, 561, 563  
   working with, 573–574  
 comparing, 134–136, 135–136*f*  
 converting to HTML, 138–141, 138–141*f*  
 custom, 305–311, 310–311*f*  
 exporting CSV (comma-separated value files), 127–132,  
   128–132*f*, 136–138, 137*f*  
 formatting  
   Format-Custom cmdlet, 141  
   Format-List cmdlet, 141, 143–144, 143–144*f*, 251  
   Format-Table cmdlet, 141–146, 144–146*f*, 313, 608, 610,  
   674  
   Format-Wide cmdlet, 141, 146–147, 147*f*  
 formatting for reports, 141–143, 142*f*  
 output creation  
   creating custom views, 313–322, 313–322*f*  
   custom object creation, 305–311, 306–311*f*  
   custom object use, 301  
   outputs designed for the pipeline, 301–305, 301–305*f*  
   properties and methods, 76–78, 78*f*, 311–313, 311–313*f*  
 piping, 148, 663  
 piping from cmdlet to cmdlet, 88–97, 88–97*f*  
 properties, 499, 561  
 Regex, 445, 539–542, 539–542*f*, 544, 547–548, 547*f*  
 serialization, 617–619, 617–619*f*  
   defined, 617–619, 617–619*f*  
   exporting XML files, 617–620*f*, 617–621  
   importing XML files, 618–619, 619*f*  
   remote management, 621–623, 622–623*f*  
   running jobs, 621, 621*f*  
   workflow, 624–625, 624–625*f*  
@ operator, 191  
operators  
 assignment, 551–552*t*, 551–557, 552–554*f*, 556–557*f*  
 associative array creation, 191, 191*f*  
 comparison, 116–120, 116–120*f*  
 logical, 120–122, 121*f*, 453  
 special, 349  
ordered hash tables, 308  
Out-File cmdlet, 439–442, 440*f*, 444  
Out-GridView cmdlet, 148–150*f*, 148–151  
Out-Null cmdlet, 660–661  
output  
 comparison operators, 117–118*f*, 117–119  
 filtering, 123–124, 123–124*f*  
 filtering data, 115–116, 115–116*f*  
 logical operators, 121, 121*f*  
 text comparison only, 119–120, 119–120*f*  
Where-Object cmdlet, 122–123, 122–123*f*

ValidateNotNullOrEmpty, 291–292, 291–292*f*  
 ValidateRange, 292–293, 293*f*  
 ValueFromPipeline, 290–291, 291*f*  
 -Binding ByPropertyName, 92–94, 92–94*f*  
 -Binding ByValue, 90–91, 90–91*f*  
 -CimSession, 414–416  
 -Credential, 450  
 datatype specification, 283  
 -Debug, 338, 352  
 -Delimiter, 132  
 -Detailed parameter switches, 41–42, 41–42*f*  
 -DontShow, 293–294  
 -ErrorAction, 346–350, 346–350*f*  
 -ErrorVariable, 346–350, 346–350*f*  
 -ExpandProperty, 81, 81*f*  
 -GroupBy, 148  
 help, 333–335, 333–335*f*  
 -JobName, 669, 670*f*  
 mandatory attributes, 289–290  
 mismatched, 94–97, 94–97*f*  
 names, help, 47, 47*f*  
 names, shortening, 47, 47*f*  
 -PassThru, 150  
 positional, 46, 46*f*  
 scripting, 281–283, 282–283*f*  
 sets, 47–48, 48*f*  
 switches, 41–42, 41–42*f*  
 syntax, 48–50, 48–50*f*  
parameter sets, 47–48, 47*f*  
parameter switches, 41–42, 41–42*f*  
parameter syntax, 48–50, 48–50*f*  
parenthesis, 98–100, 98–100*f*  
parentheticals, 97–100  
parsing  
  INI files, 446–448  
  ISS log files, 443–445  
  text files, 443, 443*f*  
  XML files, 657  
-*PassThru* parameter, 150  
passwords, 492, 580–581, 601  
path cmdlets  
  Convert-Path, 457–458, 458*f*  
  Join-Path, 437  
  Resolve-Path, 460–462, 461*f*  
  Split-Path, 458–460, 459–460*f*  
  Test-Path, 457, 457*f*  
-*Path* parameter, 45, 64, 72  
PauseService() method, 491  
permissions  
 automating Cacls.exe to change permissions, 472–473  
 changing, 470–471  
 complex permissions in PowerShell  
  Get-Owner cmdlet, 473–475, 474–475*f*  
  SetOwner() method, 475  
managing, 465  
removing a rule, 478  
retrieving access control, 476–478, 476*f*  
viewing, 465–469, 466–469*f*  
viewing for an entire hierarchy, 469–470, 469–470*f*  
Ping-computer function, 635  
pipelines  
 executing methods on an object, 81–85, 82–85*f*  
 ForEach-Object cmdlet, 81–85, 82–85*f*, 204, 213  
 information selection, 78–81, 78–81*f*

**P**

Parallel keyword, 679  
parameters  
 -AsJob parameter, 669, 670*f*  
 attributes, 289–294, 290–293*f*  
  DontShow, 293–294  
  mandatory, 289–290, 290*f*

- object types, methods, and properties, 76–78, 78f  
 outputs designed for the pipeline, 301–305, 302–305f  
 parameters  
   -ByPropertyName, 92–94f  
   -ByValue, 90–91, 90–91f  
   -ExpandProperty, 81, 81f  
   mismatched, 94–97, 94–97f  
   parentheses, 98–100, 98–100f  
   parentheticals, 97–100  
 purpose, 75–76, 76f, 88–89, 88–89f  
 sorting and measuring data for output, 103–112  
 sorting your results, 103–105f, 103–106  
 positional parameters, 46, 46f  
**PowerShell**  
   aliases, 26–29  
   color tab, 11  
   core cmdlets, 25–26, 25–26f  
   defined, 1–2  
   familiar commands, 23–25, 24f  
   font tab, 11  
   future of, 3  
   ISE configuration, 10, 12–13, 12f, 153  
   lab environment, 4–5  
   launching, 9  
   layout tab, 11  
   learning guide, 3  
   management capabilities  
     adding snap-ins, 169–170, 170f  
     extended, 167–169, 168f  
     importing modules, 169, 170–172, 171–172f, 294–295, 295f  
   options tab, 11  
   reference guide, 4  
   remoting, 236–237, 236–237f  
   text based console, 10  
   tools, 5  
   versions and requirements, 7–8  
   visual integrated scripting environment, 10  
 Windows Management Framework installation, 8–9, 235–236  
 workflow, 667–681, 668–681f  
**PowerShell hosts**  
   Culture, 641–642, 641f  
   \$host variable, 639–641, 640–641f, 649–650  
   nested prompts, 647–648, 648f  
   prompting user to make choice, 649–650  
   quitting PowerShell, 649  
   UI and RawUI use  
     changing colors, 645–646, 646f  
     changing the window title, 644–645  
     changing window size and buffer, 646–647, 646f  
     reading lines and keys, 642–643, 643f  
**PowerShell remoting**, 236–237, 236–237f  
**PowerShell Studio**  
   components, 19–20, 19f  
   configuration, 13, 18–21  
   help, 50–51, 50–51f  
   installation, 18  
   interface, 18, 18f  
   launching, 19–20  
   scripting, 20–21, 21f, 153–154  
   scripting security, 162–163, 163f  
   setting fonts, 20, 20f  
**PrimalScript 2014**
- components, 14–15, 14f  
 configuration, 13–18  
 files and fonts, 15–17, 16–17f  
 help, 50–51, 50–51f  
 installation, 13  
 scripting, 17–18, 18f, 153  
 scripting security, 162, 162f  
**Process block**, 358  
**processes**  
   detailed process information, 495–496, 495–496f  
   find process details, 499–500, 500f  
   find process owners, 500–502, 501–502f  
   local, 497–498  
   longest running, 498–499  
   managing, 493–495, 493–495f  
   memory usage, 498  
   process-management tasks, 498–499  
   remote, 502–505, 503–504f  
   remote process creation, 505–506  
   starting a process, 497  
   stopping a remote process, 506  
   stopping local processes, 497  
   tasks, 498–499  
   waiting on a process, 498–499  
**properties**, 2, 163, 369, 372, 501  
**Properties extension**, 77  
**-Property parameter**, 388  
**providers and drives**  
   capabilities, 58–59, 58–59f  
   changing location, 68–69, 68–69f  
   Get-PSDrive cmdlet, 55, 56, 56f, 57f  
   item manipulation cmdlets, 70–72, 70–72f  
   listing child items, 63–67, 64–66f  
   managing, 60  
   mapping drives, 60–63, 61–63f  
   New-PSDrive cmdlet, 55, 62  
   New-SMBMapping cmdlet, 62–63, 63f  
   provider help, 59–60, 60f  
   PSDrives, 55–57, 56f  
   PSProviders, 57–58, 58f  
   Remove-PSDrive cmdlet, 55, 61  
**PSCustomObject cmdlet**, 310–311  
**PSCustomObject type accelerator**, 310, 316  
**PSDesiredStateConfigurations cmdlets**, 403  
**PSDrives**, 55–57, 56f, 60, 508, 517, 628–630  
**PSProviders**, 57–58, 58f  
**PSScheduledJob module**, 270–271, 270f  
**PSSessions**, 244, 246

**Q**

- queries, 560, 588, 601, 657  
 quotation marks, 46, 72

**R**

- Rd command, 24  
 RDP (Remote Desktop Protocol), 13, 235  
 Read-Host cmdlet, 472, 642–643  
 ReadKey() method, 643  
 ReadLine() method, 643  
 Receive-Job cmdlet, 263–264f, 267f, 277, 673

Receive-PSSession cmdlet, 246, 246*f*  
 -Recurse parameter, 71, 469  
 Regex object, 445, 539–542, 539–542*f*, 544  
 Register-ScheduledJob cmdlet, 275–277, 275–277*f*  
 Register-WmiEvent cmdlet, 610–611, 610*f*, 613–614  
 registries  
   creating registry items, 512–514, 513–514*f*  
   managing, 507–512, 508–512*f*  
   managing remote registries with WMI, 517–518, 518*f*  
     enumerating keys, 518  
     enumerating values, 518–522  
     modifying, 522–523  
       with the .Net Framework, 523–524  
   registry rules, 514  
   removing registry items, 514  
   searching, 505–516*f*, 505–517, 515–517  
 regular expressions  
   about, 525–530  
   e-mail address matches, 546–547, 547*f*  
   groupings, 537*t*  
   IP addresses, 533–534, 533–534*f*  
   matching whitespaces, 548–549, 548*f*  
   named character sets, 534–535*f*, 534–537, 536–537*t*  
   quantifiers, 526*t*  
   Regex object, 445, 539–542, 539–542*f*  
   replace operator, 542–545, 542–545*f*  
   Select-String cmdlet, 538–539, 538*f*  
   special characters, 526*t*  
   telephone numbers, 549  
   UNC path string, 531–532*f*, 531–533  
   Unicode character sets, 536–537*t*  
   writing, 527–530, 527–530*f*  
 remote background jobs, 261–263, 261–263*f*  
 remote computers, 491  
 Remote Desktop Protocol (RDP), 13, 235  
 remote event logs, 430–431  
 remote management, 621–623, 621–623*f*  
 Remote Server Administration Tools (RSAT), 370  
 RemoteSigned execution policy, 156–157, 157*f*, 165  
 remoting  
   advanced session configurations, 248–252, 249–252*f*  
   controlling services, 491  
   disconnected sessions, 244–246, 244–246*f*  
   enabling PowerShell remoting, 236–237, 236–237*f*  
   establishing sessions, 242–244, 242–244*f*  
   implicit remoting, 246–248, 246–248*f*  
   managing-one-to-many, 239–241, 239–241*f*  
   managing-one-to-one, 238, 238*f*  
   multi-hop remoting, 252–256, 253–256*f*  
   remote command output, 241, 241*f*  
   Windows Management Framework, 8–9, 235–236  
   WS-MAN protocol, 236  
 Remove-Alias cmdlet, 630  
 Remove-Event cmdlet, 613  
 Remove-Item cmdlet, 450, 463, 463*f*  
 Remove-ItemProperty cmdlet, 514  
 Remove-Job cmdlet, 267, 267*f*  
 Remove-PSBreakPoint cmdlet, 345  
 Remove-PSDrive cmdlet, 55, 61  
 Remove-PSSession cmdlet, 244, 244*f*  
 Remove-PSSnapin cmdlet, 169–170, 170*f*  
 Remove-Variable cmdlet, 183  
 Rename-Item cmdlet, 451–452  
 replace operators, 542–545, 542–545*f*

Resolve-Path cmdlet, 460–462, 461*f*  
 Restart-Service cmdlet, 486–487, 487*f*  
 Restore-DscConfiguration cmdlet, 417  
 Restricted execution policy, 156  
 Resume-Job cmdlet, 673  
 Resume-Service cmdlet, 486  
 ResumeService() method, 491

## S

Saved Queries, 588  
 scheduled jobs  
   function, 270–271, 270*f*  
   options, 271–273, 271–273*f*  
   triggers, 274–275, 274–275*f*  
 scopes  
   best practices, 631–633, 632*f*  
   cmdlets, 630  
   dot sourcing, 634–635, 634–635*f*  
   forcing best practices, 633–634  
   nested prompts, 636, 636*f*  
   parameters, 631  
   rules, 628–630, 628–630*f*  
   scope-aware elements, 628  
   special scope modifiers, 638  
   specifying, 630  
   tracing complicated nesting scopes, 637  
   types of, 627  
 script blocks, 504  
 scripting  
   adding a manifest for your module, 295–296*f*, 295–298, 298*f*  
   Break statement, 210–211, 210–211*f*  
   commands, 178  
   Continue statement, 210–211, 211*f*  
   Do/Until loop, 208–210, 209–210*f*  
   Do/While loop, 208, 208*f*  
   Foreach loop, 204–206, 205–206*f*, 312, 472–473  
   For loop, 206–207, 207*f*  
   function into advanced function, 287–289, 288–289*f*  
   function into module, 294–295, 294–295*f*  
   (If) statement, 197–202, 198–202*f*, 203  
   parameter attributes, 289–294, 290–293*f*  
   parameter use, 281–283, 282–283*f*  
   PowerShell Studio, 20–21, 21*f*, 153–154  
   PrimalScript 2014, 17–18, 18*f*, 153  
   script into function, 283–287, 284–287*f*  
   scripts versus modules, 294  
   Switch statement, 202–204, 203*f*, 204*t*  
   variables and scope, 287  
   While loop, 207–208, 207–208*f*  
 scripting security  
   digitally signing scripts, 159–160  
   execution policies, 156–157, 160, 165  
   Internet safe scripts, 160–161, 161*f*  
   least privilege principle, 160–161  
   script files, 153–154  
   security features, 154–155, 154–155*f*  
   self-signed certificates, 159–160  
   signing with PowerShellStudio, 162–163, 163*f*  
   signing with PrimalScript, 162, 162*f*  
   trusted scripts, 159  
   turning one-liners into scripts, 163–165, 164–165*f*  
 ScriptProperty extension, 77

- scripts  
 alternative credentials, 59  
 arrays  
     associative, 190–192, 191–192*f*  
     collection, 187–189, 188*f*  
     working with, 189–192, 189–192*f*  
 comment-based help for, 332  
 debugging, 342, 342*f*  
 files, 153–154  
 new cmdlets, 25, 274, 592  
 scope, 284, 627, 634, 637  
 variables  
     Clear-Variable cmdlet, 182–183  
     Get-Variable cmdlet, 180–181, 180–181*f*  
     New-Variable cmdlet, 181  
     Remove-Variable cmdlet, 183  
     Set-Variable cmdlet, 181–182*f*, 181–183
- SearchForAllUsersAdvanced.ps1, 589  
 SearchForAllUsers.ps1, 588  
 Select-Object cmdlet, 78–81, 78–81*f*, 103, 108–110, 128–129, 309, 442, 445  
 Select-String cmdlet, 443–444, 446, 538–539, 538*f*  
 self-signed certificates, 159–160  
 ServerToolKit module, 298, 319–320  
 services  
     change service logon account, 490–491, 491*f*  
     change service logon account password, 492  
     controlling on remote computers, 491  
     elevated sessions, 482–484, 484*f*  
     listing, 479–482  
     managing, 479, 487–490  
     restarting, 486–487  
     starting, 484  
     stopping, 485, 485*f*  
     suspending and resuming, 485–486, 485–486*f*  
 Set-Acl cmdlet, 470, 472  
 Set-Alias cmdlet, 29  
 Set-Attributes() method, 454–455, 455*f*  
 Set-ExecutionPolicy cmdlet, 165  
 SetInfo() method, 562–563, 566  
 Set-ItemProperty cmdlet, 511  
 Set-JobTrigger cmdlet, 275  
 Set-Location cmdlet, 68–69, 68–69*f*, 508  
 SetOwner() method, 475  
 SetPasswords() method, 563  
 SetPermswithCACLS.ps1, 472  
 Set-PSBreakPoint cmdlet, 343  
 Set-ReadOnly() method, 452–453  
 Set-ScheduledJobOption cmdlet, 273  
 Set-Service cmdlet, 487–488  
 Set-Strict cmdlet, 633  
 Set-Variable/New-Variable cmdlet, 181–182, 182*f*  
 shortening parameter names, 47*f*  
 signing scripts, 162–163, 162*f*, 163*f*  
 snap-in cmdlets, 169–170, 170*f*  
 sorting and measuring data for output  
     case sensitivity, 107, 107*f*  
     displaying unique information, 106–107, 106–107*f*  
     measuring results, 110–111*f*, 110–112  
     select and sort a subset of data, 108–110, 108–110*f*  
     sorting results, 103–105*f*, 103–106  
 Sort-Object cmdlet, 103–110, 103–110*f*, 426  
 Splatting, 356–357  
 Split() method, 446  
 Split-Path cmdlet, 458–460, 459–460*f*  
 SQL Server, 170, 213  
 Start-Job cmdlet, 260–261, 260–261*f*  
 -StartPassword parameter, 492  
 Start-Service cmdlet, 484, 486  
 Start-Sleep cmdlet, 260  
 Start-Transcript cmdlet, 32–33, 32–33*f*  
 StdReg provider, 517  
 Stop-Process cmdlet, 92–94, 92–94*f*, 161, 161*f*, 494–495, 494–495*f*, 497, 497*f*  
 Stop-Service cmdlet, 241, 485–486, 485*f*  
 Stop-Transcript cmdlet, 32–33, 32–33*f*  
 string operators, 548–549, 548*f*  
 subexpressions, 193–194, 194*f*  
 Suspend-Job cmdlet, 671–672  
 Suspend-Workflow cmdlet, 672  
 -Switch parameter, 364  
 Switch statement, 202–204, 203*f*, 204*f*
- T**  
 tab completion, 31–32, 44, 47, 129, 508  
 Tab key, 508  
 Tee-Object cmdlet, 362  
 Test-DscConfiguration cmdlet, 415–416  
 Test-Path cmdlet, 457, 457*f*  
 Test-Port functions, 321  
 text boxes, 375, 389–391  
 text comparison, 119–120, 119–120*f*  
 text files, 439–441*f*, 439–443  
 TFM.System.Inventory object, 317–318, 318*f*  
 topic oriented help files, 42–44, 43–44*f*  
 ToString() method, 446  
 Total-Count cmdlet, 441  
 transcript cmdlets, 32–33, 32–33*f*  
 Trap keyword, 355  
 trusted scripts, 159  
 Try-Catch-Finally block, 355–360, 355–360*f*  
 type adapters, 576–580, 577*f*, 579–580*f*
- U**  
 Uncompress() method, 456  
 UNC path string, 531–532*f*, 531–533  
 Undefined execution policy, 156  
 Unicode character sets, 536–537*f*  
 unique information, 106–107, 106–107*f*  
 Unregister-Event cmdlet, 613  
 Unregister-ScheduledJob cmdlet, 278  
 Unrestricted execution policies, 156  
 Update-Help cmdlet, 38–39, 39*f*  
 user interfaces, 2, 369, 642, 649  
 UserName text box, 389  
 users, directories, 2, 581–584, 587
- V**  
 ValidateNotNullOrEmpty parameter, 291–292, 291–292*f*  
 ValidateRange parameter, 292–293, 293*f*  
 ValueFromPipeline parameter, 290–291, 291*f*  
 variables

- creating with cmdlets, 178–183, 180–182/*f*  
defining, 175–178, 176–177/*f*  
inside quotes, 192–194/*f*, 192–195  
with multiple objects, 187–192, 188–192/*f*  
\$MyVar, 175–178, 177–178/*f*  
as objects, 183–187, 184–186/*f*  
scripts  
  Clear-Variable cmdlet, 182–183  
  Get-Variable cmdlet, 180–181, 180–181/*f*  
  New-Variable cmdlet, 181  
  Remove-Variable cmdlet, 183  
  Set-Variable cmdlet, 181–182/*f*, 181–183  
subexpressions, 193–194, 194/*f*  
\$var, 636–637
- VBScript, 154, 215–216
- VBScript alert, 601–602
- Verbose parameter, 276, 353, 405, 406/*f*, 412, 416
- W**
- Wait parameter, 412
- Wait-Process cmdlet, 498
- watching services, 614–615
- WhatIf parameter, 94, 94/*f*, 161, 161/*f*
- Where-Object cmdlet, 115–116/*f*, 115–117, 122–123, 122–123/*f*, 423, 495
- While statement, 207–208, 207–208/*f*
- Windows Management Framework installation, 8–9, 235–236
- Windows Remote Management, 8, 236
- Windows Server 2008 R2, 4–5, 13, 168, 235, 405, 429, 482, 575
- Windows servers, 236, 479
- Windows Workflow Foundation (WWF), 669
- WinForms tools creation
- API use, 369
  - common controls, 371–378, 372–378/*f*
  - events and controls, 379–401, 379–401/*f*
  - PowerShell example, 369–371, 370–371/*f*
- WMI (Windows Management Instrumentation)
- events
    - actions, 610–613, 610–613/*f*
    - practical uses, 607–610, 609/*f*
    - querying for specific events, 613–614
  - invoking WMI methods, 606–607
  - managing remote registries
    - enumerating keys, 518
    - enumerating values, 518–522
    - modifying, 522–523
      - with the .NET Framework, 523–524
  - remote management, 600–601, 601/*f*
  - retrieving information, 600/*f*
  - systems management
    - cmdlets, 593–595, 594/*f*
    - examining existing values, 598–599
    - listing available classes, 596–597, 597/*f*
    - listing properties of a class, 597–598, 598/*f*
    - retrieving basic information, 595–596, 595–596/*f*
- VBScript alert, 601–602
- watching files and folders, 615–616
- watching services, 614–615
- WMI/CIM
- CIM cmdlets, 215/*f*
  - classes, 217–220, 217–220/*f*
  - executing WMI/CIM methods, 222–224, 222–224/*f*
- Get-WMIObject versus Get-CIMInstance, 214–215, 214–215/*f*
- methods, 222–224, 222–224/*f*
- namespaces, 213
- navigating and using, 215–216, 216/*f*
- objects and properties, 220–222, 221–222/*f*
- remote computer reach, 225, 225/*f*
- security and end points with WMI, 226–232
- structure, 213
- [WMISearcher] type, 605–606, 605–606/*f*
- [WMI] type, 603–605, 603–605/*f*
- WMI Explorer, 5, 216, 216/*f*, 301–302, 599–600, 616
- WMI queries, 601
- WMIReport.ps1, 602
- [WMISearcher] type, 605–606, 605–606/*f*
- [WMI] type, 603–605, 603–605/*f*
- workflow activities, 624–625, 624–625/*f*, 667–681, 668–681/*f*
- Workflow keyword, 667–668
- working with individual elements in an array, 189–192, 189–192/*f*
- Write-Debug cmdlet, 337–339, 337–339/*f*, 350–354, 350–354/*f*
- Write-Debug statement, 338
- Write-Error cmdlet, 360–362, 360–362/*f*
- Write-Host cmdlet, 608, 643
- Write-Output cmdlet, 175
- Write-Verbose statement, 335–336, 335–336/*f*, 350–354, 350–354/*f*
- Write-Warning cmdlet, 360–362, 360–362/*f*
- WS-MAN protocol, 236
- WWF (Windows Workflow Foundation), 669

**X**

- XAML (Extensible Application Markup Language), 668–669, 669/*f*
- Xcacls.exe command-line utility, 465
- XML files
- converting objects into, 136–138, 137/*f*, 651–652
  - exporting, 617–621, 618–620/*f*
  - filters, 429
  - importing, 132–134, 133/*f*, 618–619, 619/*f*
  - manipulation, 653–657, 653–657/*f*
  - practical example, 657–663
  - turning objects into, 663–665, 663–665/*f*
- XMLInventory.ps1, 661–662
- XML objects, 241
- XPath queries, 657





Join the leading edge of Windows administrative scripting with Windows PowerShell, the comprehensive command-line environment and scripting shell for Windows and Windows-based server products. PowerShell offers that fastest, easiest, and most consistent way to automate almost any administrative task, including IIS, Active Directory, SQL Server, Exchange, System Center, Remote Desktop Services, and much more. This all-new *Windows PowerShell: TFM* is fully updated to cover all new and improved PowerShell v4 features. If you can do it with Windows PowerShell, it's covered here, starting with absolute beginners all the way through advanced topics. No other book on Windows PowerShell provides this much coverage, with such clear explanations, from two recipients of Microsoft's Most Valuable Professional (MVP) Award!



**Jason C. Helmick** is Senior Technologist and a Windows PowerShell MVP. His IT career spans more than 25 years enterprise consulting on a variety of technologies, with a focus on strategic IT business planning. Jason serves as board member and COO/CFO of PowerShell.org.



**Mike F. Robbins** is a PowerShell MVP, IT pro, and winner of the advanced category in the 2013 scripting games. He has written guest blog articles for the Hey, Scripting Guy! Blog, PowerShell.org, PowerShell Magazine, and is a contributing author of a chapter in the PowerShell Deep Dives book. Mike is also the leader and co-founder of the Mississippi PowerShell User Group. He blogs at <http://mikefrobbins.com> and can be found on twitter @mikefrobbins.

USA \$ 59.99  
CAD \$ 69.99  
GBP £ 39.99  
EURO € 49.99

\$ 59 . 99  
ISBN 978-0-9821314-6-6  
5 5 9 9 9 >



9 780982 131466



SAPIEN Press  
A division of SAPIEN Technologies, Inc.  
831 Latour Court, Suite B2  
Napa, CA 94558 USA  
1.866.774.6257 | [www.sapien.com](http://www.sapien.com)