# *Learn* WINDOWS POWERSHELL IN A MONTH OF LUNCHES


DON JONES

MANNING

*Learn Windows PowerShell*
*in a Month of Lunches*

# *Learn Windows PowerShell in a Month of Lunches*

DON JONES

MANNING
Shelter Island

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

©2011 by Manning Publications Co. All rights reserved.

® In a Month of Lunches is a registered trademark of Don Jones.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without elemental chlorine.

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Maria Townsley
Copyeditor: Andy Carroll
Proofreader: Katie Tennant
Typesetter: Marija Tudor
Cover designer: Leslie Haimes

ISBN: 9781617290213
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 17 16 15 14 13 12 11

*For Chris … for another fifteen years and more*

# brief contents

# contents

# *preface*

I've been teaching and writing about Windows PowerShell for a long time. As I began contemplating this book, I realized that most PowerShell writers and teachers—including myself—were forcing our students to approach the shell as a kind of programming language. Most PowerShell books are into "scripting" by the third or fourth chapter, yet more and more PowerShell students were backing away from that programming-oriented approach. Those students wanted to use the shell as a shell, at least at first, and we simply weren't delivering a learning experience that matched that desire.

So I decided to take a swing at it. A blog post on WindowsITPro.com proposed a table of contents for this book, and ample feedback from the blog's readers fine-tuned it into the book you're about to read. I wanted to keep each chapter short, focused, and easy to cover in a short period of time—because I know administrators don't have a lot of free time, and often have to learn on the fly.

I also wanted a book that would focus on PowerShell itself, and not on the myriad technologies that PowerShell touches, like Exchange Server, SQL Server, System Center, and so on. I truly feel that by learning to use the shell properly, you can teach yourself to administer all of those "PowerShell-ed" server products. So this book tries to focus on the core of using PowerShell. Even if you're also using a "cookbook" style of book, which provides ready-to-use answers for specific administrative tasks, this book will help you understand what those examples are doing. That understanding will make it easier to modify those examples for other purposes, and eventually to construct your own commands and scripts from scratch.

I hope this book won't be the only PowerShell education that you pursue. In fact, this book's companion website, MoreLunches.com, is designed to help you continue that education in small chunks. It offers free videos that correspond to this book's chapters, letting you see and hear my demonstrations of key techniques. I'll also be posting supplemental articles, and recommending additional resources for you to investigate.

If you happen to run into me at a conference—I'm a regular at Windows Connections, TechMentor events, and Microsoft TechEd—I hope you'll come up and say hello. Let me know how this book is working for you, and what other resources you've found useful. You can also contact me via email through ConcentratedTech.com, or on manning.com in this book's discussion forum.

Enjoy—and good luck with the shell.

# about this book

Most of what you'll need to know about this book is covered in chapter 1, but there are a few things that we should mention up front.

First of all, if you plan to follow along with my examples and complete the hands-on exercises, you'll need a virtual machine or computer running Windows Server 2008 R2. I cover that in more detail in chapter 1.

Second, be prepared to read this book from start to finish, covering each chapter in order. Again, this is something I'll explain in more detail in chapter 1, but the idea is that each chapter introduces a few new things that you will need in subsequent chapters.

Third, this book contains a lot of code snippets. Most of them are quite short, so you should be able to type them quite easily. In fact, I recommend that you do type them, since doing so will help reinforce an essential PowerShell skill: accurate typing! Longer code snippets are given in listings and are available for download at http://MoreLunches.com or from the publisher's website at www.manning.com/LearnWindowsPowerShellinaMonthofLunches.

That said, there are a few conventions that you should be aware of. Code will always appear in a special font, just like this example:

```
Get-WmiObject -class Win32_OperatingSystem
➥ -computerName SERVER-R2
```

That example also illustrates the line-continuation character used in this book. It indicates that those two lines should actually be typed as a single line in PowerShell. In other words, don't hit Enter or Return after `Win32_OperatingSystem`—keep right on typing. PowerShell allows for very long lines, but the pages of this book can only hold so much.

Sometimes, you'll also see that code font within the text itself, such as when I write `Get-Command`. That just lets you know that you're looking at a command, parameter, or other element that you would actually type within the shell.

Fourth is a tricky topic that I'll bring up again in several chapters: the backtick character (`). Here's an example:

```
Invoke-Command -scriptblock { Dir } `
-computerName SERVER-R2,localhost
```

The character at the end of the first line isn't a stray bit of ink—it's a real character that you would type. On a U.S. keyboard, the backtick (or grave accent) is usually near the upper left, under the Escape key, on the same key as the tilde character (~). When you see the backtick in a code listing, type it exactly as is. Furthermore, when it appears at the end of a line—as in the preceding example—make sure that it's the very last character on that line. If you allow any spaces or tabs to appear after it, the backtick won't work correctly, and neither will the code example.

Finally, I'll occasionally direct you to internet resources. Where those URLs are particularly long and difficult to type, I've replaced them with Manning-based shortened URLs that look like http://mng.bz/S085 (you'll see that one in chapter 1).

## Author Online

The purchase of *Learn Windows PowerShell in a Month of Lunches* includes access to a private forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and other users. To access and subscribe to the forum, point your browser to www.manning.com/LearnWindowsPowerShellinaMonthofLunches, and click the Author Online link. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct in the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

# *about the author*

Don Jones is a multiple-year recipient of Microsoft's prestigious Most Valuable Professional (MVP) Award for his work with Windows PowerShell. He writes the Windows PowerShell column for Microsoft TechNet Magazine, the *PowerShell with a Purpose Blog* for WindowsITPro.com, and the "Decision Maker" column for *Redmond Magazine*. Don is a prolific technology author and has published more than a dozen print books since 2001. He has also authored numerous free ebooks for RealtimePublishers.com and currently serves as that company's Editor-in-Chief and CTO. Don is a Senior Partner and Principal Technologist for Concentrated Technology (ConcentratedTech.com), an IT education and strategic consulting firm. Don's first Windows scripting language was KiXtart, all the way back in the mid-1990s. He quickly graduated to VBScript in 1995 and was one of the first IT pros to start using early releases of a new Microsoft product code-named "Monad"—which later became Windows PowerShell. Don lives in Las Vegas and travels all over the world delivering IT training (especially in PowerShell) and speaking at IT conferences.

# *acknowledgments*

Books simply don't write, edit, and publish themselves. I'd like to thank everyone at Manning Publications who decided to take a chance on a very different kind of book for Windows PowerShell, and who worked so hard to make it happen.

I'd also like to acknowledge everyone who provided feedback for this book, starting with the simple blog post on WindowsITPro.com that got the table of contents rolling, and continuing through all the Manning Early Access Program (MEAP) readers and the outside manuscript reviewers, including Ray Booysen, Margriet Bruggeman, Nikander Bruggeman, Chuck Durfee, David Moravec, and Dave Pawson. Special thanks to Richard Siddaway for his final technical review of the manuscript during production.

# 1 Before you begin

I've been teaching Windows PowerShell since version 1 was released in 2006. Back then, most of the folks using the shell were pretty experienced VBScript users, and they were eager to apply their VBScript skills to learning PowerShell. As a result, I and the other folks who taught the shell, wrote books and articles, and so forth, all adopted a teaching style that more or less leveraged prior programming or scripting skills.

Since late 2009, however, a shift has occurred. More and more administrators who *didn't* have prior VBScript experience started trying to learn the shell. All of a sudden, my old teaching patterns didn't work very well, because I was focused on scripting and programming. That's when I realized that PowerShell isn't really a scripting language. It's really a command-line shell where you run command-line utilities. Like all good shells, it has scripting capabilities, but you don't have to use them, and you certainly don't have to start with them. I started changing my teaching patterns, beginning with the many conferences I speak at each year, and moving into the instructor-led training courseware that I'd written.

This book is the result of that process, and it's the best way that I've yet devised to teach PowerShell to someone who might not have a scripting background (although it certainly doesn't hurt if you do). But before we jump into the actual instruction, let me set the stage for you.

## 1.1 Why you can't afford to ignore PowerShell

Batch. KiXtart. VBScript. Let's face it; Windows PowerShell isn't exactly Microsoft's (or anyone else's) first effort at providing automation capabilities to Windows administrators. I think it's valuable to understand why you should care about

PowerShell, so that you can feel comfortable that the time you'll commit to learning it will pay off for you. Let's start by considering what life was like before PowerShell came along, and look at some of the advantages now that we have our new shell.

### LIFE WITHOUT POWERSHELL

Windows administrators have always been happy clicking around in the graphical user interface (GUI) to accomplish their chores. After all, the GUI is pretty much the whole point of Windows—the operating system isn't called "Text," after all! GUIs are great because they enable you to discover what you can do. I remember the first time I opened Active Directory Users and Computers: I hovered over icons and read tooltips, I pulled down menus, and I right-clicked things, all to see what was available. GUIs definitely make learning a tool easier. Unfortunately, GUIs have zero return on that investment. If it takes you five minutes to create a new user in Active Directory (and assuming you're filling in a lot of the fields, that's pretty reasonable), you'll never get any faster than that. A hundred users will take five hundred minutes—there's no way, short of learning to type and click a bit faster, to make the process go any quicker.

Microsoft has tried to deal with that problem a bit haphazardly, and VBScript was probably the most successful attempt. It might have taken you an hour to write a VBScript that could import new users from a CSV file, but once you'd invested that hour, creating users in the future would only take a few seconds. The problem with VBScript is that it wasn't a wholehearted effort on Microsoft's part. Microsoft had to remember to make things VBScript-accessible, and when they forgot (or didn't have time), you were stuck. Want to change the IP address of a network adapter using VBScript? Okay, you can. Want to check its link speed? You can't, because nobody remembered to hook that up in a way that VBScript could get to. Sorry. Jeffrey Snover, the architect of Windows PowerShell, calls this "the last mile": you can do a lot with VBScript (and other, similar technologies), but it tends to always let you down at some point, never getting you through that "last mile" to the finish line.

Windows PowerShell is an express attempt on Microsoft's part to do a better job, and to get you through the last mile.

### LIFE WITH POWERSHELL

The goal behind Windows PowerShell is that Microsoft builds 100 percent of a product's administrative functionality in the shell. They continue to build GUI consoles, but those consoles are executing PowerShell commands behind the scenes. That approach forces them to make sure that every possible thing that you can do with the product is accessible through the shell. If you need to automate a repetitive task or create a process that the GUI doesn't enable well, you can drop into the shell and take full control for yourself.

A number of Microsoft products have already adopted this approach, including Exchange Server 2007 and 2010, SharePoint Server 2010, many of the System Center products, and many components of Windows itself. Going forward, more and more products and Windows components will follow this pattern. That's exactly why you can't afford to ignore PowerShell: over the next few years, it will become the basis for more and more administration.

Ask yourself a question: if you were in charge of a team of IT administrators (and perhaps you are), which ones would you want in your senior, higher-paying positions? The ones who need several minutes to click their way through a GUI each time they need to perform a task, or the ones who can perform tasks in a few seconds after automating them? We already know the answer from almost every other part of the IT world. Ask a Cisco administrator, or an AS/400 operator, or a Unix administrator. The answer is: "I'd rather have the guy or gal who can run things more efficiently from the command line." Going forward, the Windows world will start to split into two groups: administrators who can use PowerShell, and those who can't. As I famously said at Microsoft's TechEd 2010 conference, your choice is "learn PowerShell, or would you like fries with that?"

I'm glad you've decided to learn PowerShell.

## 1.2 Is this book for you?

This book doesn't try to be all things to all people. In fact, Microsoft's PowerShell team loosely defines three audiences who use PowerShell:

- Administrators who primarily run commands and consume tools written by others.
- Administrators who combine commands and tools into more complex processes, and perhaps package those as tools that less-experienced administrators can utilize.
- Administrators and developers who create reusable tools and applications.

This book is designed primarily for the first audience, and to a lesser degree the second audience. I think it's valuable for anyone, even a developer, to understand how the shell is used to run commands. After all, if you're going to create your own tools and commands, you should know the patterns that the shell uses, so that you can make tools and commands that work as well as they can within the shell.

If you're interested in creating scripts to automate complex processes, such as new user provisioning, then you'll absolutely see how to do that by the end of this book. You'll even see how to get started on creating your own commands that other administrators can use. This book won't, however, plumb the depths of everything that PowerShell can possibly do. The goal here is to get you using the shell, and using it effectively, in a production environment.

## 1.3 How to use this book

The idea behind this book is that you'll read one chapter each day. You don't have to read it during lunch, but each chapter should only take you about 40 minutes or so to read, giving you an extra 20 minutes to gobble down the rest of your sandwich and practice what the chapter showed you.

#### THE MAIN CHAPTERS

Of the 28 chapters in this book, chapters 2 through 24 contain the main content, giving you 23 days' worth of lunches to look forward to, meaning you can look forward to completing the main content of the book in about a month. Try to stick with that schedule as much as possible, and don't feel the need to read extra chapters in a given day. It's actually more important that you spend some time practicing what each chapter shows you, because using the shell will help cement what you've learned. Not every single chapter will require a full hour, so sometimes you'll be able to spend some additional time practicing (and eating lunch) before you have to get back to work.

#### HANDS-ON LABS

Most of the main content chapters include a short lab for you to complete. You'll be given instructions, and perhaps a hint or two, but you won't find any answers in the book. The answers are online, at MoreLunches.com, but try your best to complete each lab without looking at the online answers.

#### SUPPLEMENTARY MATERIALS

The MoreLunches.com website also contains additional supplementary content, including extra chapters, companion videos, and so forth. In fact, each chapter has at least one companion video so that you can see what the chapter is describing, happening in an actual PowerShell window. The videos are only five minutes or so apiece, so you should have time to watch them when you're done reading the chapters.

#### IDEAS FOR ON YOUR OWN

Some chapters conclude with ideas for further exploration on your own. Again, try to find some spare minutes each afternoon to tackle these short challenges, because doing so will definitely improve your skill and comfort in the shell.

#### GOING FURTHER

The last four chapters in this book will help you take your newfound PowerShell skills and put them to work, take them further, and keep them fresh. Those chapters might not fit into a single hour, and they don't come with labs, but they will get you started on using PowerShell in the real world.

#### ABOVE AND BEYOND

As I learned PowerShell myself, there were often times when I wanted to go off on a tangent and explore why something worked the way it did. I didn't learn a lot of extra practical skills that way, but I did gain a deeper understanding of what the shell is, and how it works. I've included some of that tangential information throughout the book in sections labeled "Above and beyond." None of those will take you more than a couple of minutes or so to read, but if you're the type of person who likes to know why something works the way it does, they can provide some fun additional facts. If you feel that those sections might distract you from the practical stuff, just ignore them on your first read-through. You can always come back and explore them later when you've mastered the chapter's main material.

## 1.4 Setting up your lab environment

You're going to be doing a lot of practicing in Windows PowerShell throughout this book, and you'll want to have a lab environment to work in—please, please, please don't practice in your company's production environment!

I suggest that you create a virtual machine to work in. Throughout this book, I'll assume that you're running Windows Server 2008 R2, and that you've configured your server to be the sole domain controller in the company.pri domain. If you choose to use the virtual machine approach, you can use whatever virtual machine technology you wish, whether it be VMWare, Microsoft, or something else. You can get a six-month trial of Windows Server 2008 R2 at http://www.microsoft.com/windowsserver2008/en/us/trial-software.aspx; you should install it in a virtual machine and promote it to be a standalone domain controller. If you're not comfortable getting a virtual machine up and running, installing the ADDS role, or promoting a domain controller, then this book probably isn't for you. I assume that you're comfortable with these basic administrative tasks.

If you're not entirely comfortable installing a domain controller (I realize that it's a task you don't do every single day), check out Netometer's screencast tutorial of the process on Windows Server 2008: http://mng.bz/S085.

Here are a few pieces of information you'll need:

- The FQDN for your forest root domain should be `company.pri`.
- The Windows NetBIOS name for your domain should be `COMPANY`.
- You should set the forest functional level and domain functional level to the highest levels available (which will either be Windows Server 2008 or Windows Server 2008 R2).
- For Additional Domain Controller Options, select only the option to install DNS Server.
- If you receive a warning about the computer having a dynamically assigned IP address, select "Yes, the computer will use a dynamically assigned IP address." It's okay that it isn't recommended—this is just a test computer.
- You may receive a warning about a delegation creation problem; just select Yes.
- Accept the default filesystem paths.
- Provide a Restore Mode password. I recommend something easy to remember, such as `P@ssw0rd`.
- When you see it, select the check box to Reboot on Completion.

Both the tutorial and the screencast assume that you've already installed the Active Directory Domain Services role by using Server Manager, so you can do that as a first step. Open Server Manager, tell it you want to add a role, and add the Domain Services role. Once that finishes, you can begin the domain controller promotion (Dcpromo) process outlined in the two tutorials.

You can complete most of the tasks in this book by using Windows 7. But chapters 5 and 7, in particular, require a Windows Server 2008 R2 domain controller if you want to follow along with my examples and complete the hands-on labs. Chapter 10 is definitely more interesting on a server than on a client operating system.

Keep in mind that, throughout this book, I'm assuming that you will be working on a Windows Server 2008 R2 system. That's a 64-bit operating system, also referred to as an "x64" operating system. As such, it comes with two copies of Windows PowerShell and the graphically oriented Windows PowerShell ISE. In the Start menu, the 64-bit versions of these are listed as "Windows PowerShell" and "Windows PowerShell ISE." The 32-bit versions are identified by an "(x86)" in the shortcut name, and you'll also see "(x86)" in the window's title bar when running those versions.

The examples in this book are based on the 64-bit versions of PowerShell and the ISE. If you're not using those, you may sometimes get slightly different results than mine when running examples. The 32-bit versions are primarily provided for backward compatibility. For example, some shell extensions are only available in 32-bit flavors and can only be loaded into the 32-bit (or "x86") shell. Unless you need to use such an extension, I recommend using the 64-bit shell when you're on a 64-bit operating system.

## 1.5 Installing Windows PowerShell

Windows PowerShell v2 is available for all versions of Windows since Windows XP, which includes Windows Server 2003, Windows Vista, Windows Server 2008, Windows Server 2008 R2, and Windows 7. The shell is preinstalled on Win2008R2 and Win7 (and any later versions), and it must be manually installed on older versions.

If you happen to be using an older version of PowerShell, visit http://download.microsoft.com and enter "powershell 2" into the search box. Locate the correct download for your version of Windows, and install it. If you're not able to find the right download, try http://support.microsoft.com/kb/968930—that should take you to the Windows Management Framework Core package, which is what PowerShell v2 is distributed with. Again, be very careful to select the right version. "x86" refers to 32-bit packages, and "x64" refers to 64-bit packages. You won't see a download for Windows 7 or Windows Server 2008 R2, because PowerShell comes preinstalled on those versions of Windows.

Note that PowerShell requires .NET Framework v2 at a minimum, and it prefers to have the latest and greatest version of the framework that you can get. I recommend installing at least .NET Framework v3.5 SP 1 to get the maximum functionality from the shell.

Note that Windows Server 2008 came with PowerShell v1, but it isn't installed by default. You can't have both v1 and v2 installed side by side, so installing v2 will make v1 inaccessible. If you have a product that absolutely requires v1 and won't run under v2, then you may want to hold off installing v2.

Installing PowerShell v2 also installs some companion technologies, including the Windows Remote Management (WinRM) service, which you'll learn more about later in this book. PowerShell is installed as a hotfix, which means that once it's installed, it can be a bit tricky to remove. Generally speaking, you won't want to remove it. PowerShell is officially a part of the core Windows operating system, and any bug fixes or updates will come down as additional hotfixes, or even in service packs, just like any other component of Windows.

There are two components to PowerShell v2: the standard, text-based console host (PowerShell.exe) and the more visual Integrated Scripting Environment (ISE; PowerShellISE.exe). The text-based console is what I use most of the time, but you're welcome to use the ISE if you prefer. Note that the ISE isn't preinstalled on server operating systems, so if you want to use it, you'll need to go into Windows Features (using Server Manager) and manually add the ISE feature. It isn't available at all on the no-GUI Server Core installation.

Before you go any further, take a few minutes to customize the shell. If you're using the text-based console host, I strongly recommend that you change the font it uses to the Lucida fixed-width font instead of the default console font. The default font makes it very difficult to distinguish some of the special punctuation characters that PowerShell uses. Follow these steps to customize the font:

1. Click the control box (that's the PowerShell icon in the upper-left of the console window) and select Properties from the menu.
2. In the dialog box that appears, browse through the various tabs to change the font, window colors, window size and position, and so forth.

Your changes will apply to the default console, meaning they'll stick around every time you open a new window.

## 1.6 Online resources

I've mentioned the MoreLunches.com website a couple of times already, and I hope you'll find time to visit. A number of supplementary resources for this book are available there:

- Companion videos for each chapter
- Example answers for each end-of-chapter lab
- Downloadable code listings (so you don't have to type them in from the book)
- Additional articles and bonus chapters
- Links to my Windows PowerShell blog, which contains even more examples and articles
- Links to my Windows PowerShell Frequently Asked Questions (FAQ)
- Links to discussion forums, where you can ask questions or submit feedback about this book

I'm passionate about helping folks like you learn Windows PowerShell, and I try to provide as many different resources as I can. I also appreciate your feedback, because that helps me come up with ideas for new resources that I can add to the site, and ways to improve future editions of this book. You can contact me through the links on

MoreLunches.com or on my company's website, http://ConcentratedTech.com. You can also find me on Twitter under @concentrateddon.

## 1.7 Being immediately effective with PowerShell

"Immediately effective" is a phrase that I've made my primary goal for this entire book. As much as possible, I'll try to have each chapter focus on something that you could use in a real production environment, right away. That means I'll sometimes gloss over some details in the beginning, but when necessary I promise to circle back and cover those details at the right time. In many cases, I had to choose between hitting you with twenty pages of theory first, or diving right in and accomplishing something without explaining all the nuances, caveats, and details. When those choices came along, I almost always chose to dive right in, with the goal of making you *immediately effective*. But all those important details and nuances will still be explained, just at a different time in the book (or, for the really subtle details that don't impact the book's content, I may explain them in an online article on MoreLunches.com).

Okay, that's enough background. It's time to start being immediately effective. Your first lunch lesson awaits.

# 2 Running commands

Start looking at PowerShell examples on the internet, and it's easy to get the impression that PowerShell is some kind of .NET Framework–based scripting or programming language. My fellow Microsoft Most Valuable Professional (MVP) award recipients, and hundreds of other PowerShell users, are pretty serious geeks, and we like to dig deep into the shell and see what we can make it do. But almost all of us began right where this chapter starts: simply running commands. That's what we'll be doing in this chapter: not scripting, not programming, but just running commands and command-line utilities.

## 2.1 Not scripting: just running commands

PowerShell, as its name implies, is a *shell*. It's similar to the Cmd.exe command-line shell that you've probably used before, and it's even similar to the good old MS-DOS shell that shipped with the first PCs back in the 1980s. It even has a strong resemblance to the Unix shells, like Bash, from the late '80s, or even the original Unix Bourne shell, introduced in the late '70s. PowerShell is much more modern, of course, but in the end, PowerShell isn't a scripting language like VBScript or KiXtart. With those languages, as with most programming languages, you sit down in front of a text editor (even if it's Windows Notepad) and type a series of keywords to form a script. You save that file, and perhaps double-click it to test it. PowerShell *can* work like that, but that's not necessarily the main usage pattern for PowerShell, especially when you're getting started. With PowerShell, you type a command, add a few parameters to customize the command's behavior, hit Return, and immediately see your results.

Eventually, you'll get tired of typing the same command (and its parameters) over and over again, so you'll copy and paste it all into a text file. Give that file a .PS1 filename extension, and you suddenly have a "PowerShell script." Now, instead of typing the command over and over, you just run that script, and it executes whatever commands are inside. This is the same pattern that you may have used with batch files in the Cmd.exe shell, but it's typically far less complex than scripting or programming.

Don't get me wrong: you can get as complex as you need to with PowerShell. In fact, it supports the same kind of usage patterns as VBScript and other scripting or programming languages. PowerShell gives you access to the full underlying power of the .NET Framework, and I've seen PowerShell "scripts" that were practically indistinguishable from a C# program written in Visual Studio. PowerShell supports these different usage patterns because it's intended to be useful to a wide range of audiences, as I described in the previous chapter. The point is that just because it supports that level of complexity doesn't mean you have to use it at that level, and it doesn't mean you can't be extremely effective with less complexity.

Here's an analogy: You probably drive a car. If you're like me, changing the oil is the most complex mechanical task you'll ever do with your car. I'm not a car geek, and I can't rebuild the engine. I also can't do those cool high-speed J-turns that you see in the movies, and you'll never see me driving a car on a "closed course" in a car commercial. But the fact that I'm not a professional stunt driver doesn't stop me from being an extremely effective driver at a less complex level. Someday I might decide to take up stunt driving for a hobby (I'm sure my insurance company will be thrilled), and at that point I'll need to learn a bit more about how my car works, master some new skills, and so on. That option is always there for me to grow into. But for now, I'm very happy with what I can accomplish as a normal driver.

For now, we're going to stick with being normal "PowerShell drivers," operating the shell at a lower level of complexity. Believe it or not, we're the primary target audience for PowerShell, so you'll find that there's an awful lot of incredible stuff that you can do. You just need to master the ability to run commands within the shell, and you're on your way.

## 2.2 Opening PowerShell

This is a good time to get PowerShell up and running, if you haven't done so already. You can use either the ISE or the regular console host. You'll find the icons for PowerShell and the ISE located on the Start menu, under Accessories.

It's *very important* that you run PowerShell as an Administrator. On Windows Vista and later versions of Windows, User Account Control (UAC) is enabled by default, and it prevents you from running programs as Administrator without taking a special step: right-click the program icon in the Start menu, and select Run as Administrator from the context menu. You need to do that every time you open the application. You can also right-click the icon, select Properties, and modify the program's properties to always run as Administrator. I find that to be more convenient, because from then on I can just click the icon in the Start menu to open the shell.

Why do you have to start the shell this way? Because although Windows PowerShell is constrained by UAC, I like to say that is doesn't participate in UAC. That is, if you try to perform a privileged operation in a non-elevated shell, you will get an "Access Denied" error message. You won't get the friendly UAC pop-up dialog box that asks you if you want to perform the operation. PowerShell isn't capable of elevating its privilege on the fly, so you have to do so each time you open a shell window.

**TRY IT NOW** Get PowerShell, or the ISE, up and running as Administrator.

If you choose to use the ISE, you'll find yourself looking at three panes within the window, as shown in figure 2.1 (I've rearranged them from the default to make them a bit easier to see and describe). The only two of these you need to care about for now are the command input pane and the output pane—you can ignore the script editor pane (shown on the right). Consider arranging the panes so that only those two are showing—you'll find buttons on the right side of the toolbar that let you reconfigure the panes, and the panes themselves can be resized by dragging the separator between them. Experiment with the pane arrangement until you're happy with it.

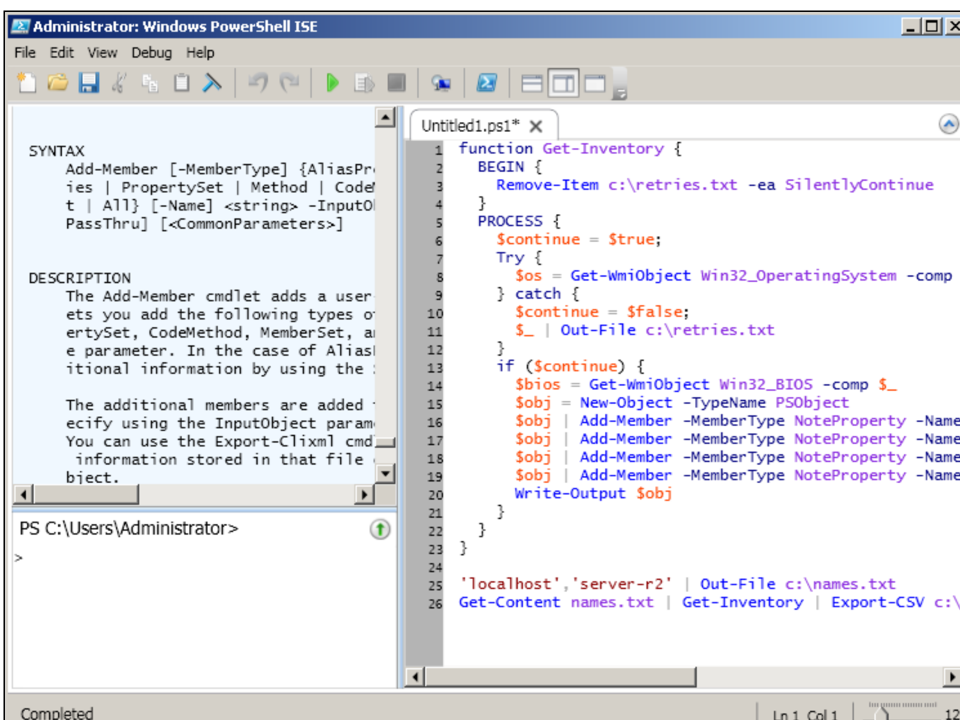



**Figure 2.1 The ISE defaults to a three-pane layout including command input, output, and script editing.**

## 2.3 Managing files and folders—you know this!

Let's start with something that you probably already know how to do: manage files and folders from the command line. Forget about PowerShell for a few minutes, and think about your existing command-line experience (if you don't have any, that's okay—you'll pick it up pretty quickly). How would you get a listing of files and folders from the current folder?

```
Dir
```

Right? Try the same command in Windows PowerShell and you'll find that it does the same thing, although the output might look a little different than it does in Cmd.exe.

> **TRY IT NOW** Try running `Dir` from within PowerShell, right now. I'd like you to get into the habit of trying what you're reading about, so that you can start getting your hands-on time with the shell as soon as possible.

Take a few minutes and see if you can remember the commands needed to use Cmd.exe to accomplish these tasks listed in table 2.1.

**Table 2.1 What commands would you use to complete these tasks in Cmd.exe?**

| Task | Cmd.exe command |
|---|---|
| Copy a file to a different location | |
| Change directories | |
| Move a file to a new location | |
| Rename an existing file | |
| Create a new directory | |
| Remove an empty directory | |
| Delete a file | |
| Display the contents of a text file | |

> **TRY IT NOW** Try running some of those same commands in PowerShell, and you should find that they all work more or less the same way that you're used to. You should have come up with the commands shown in table 2.2.

**Table 2.2 Old-style commands for file and folder management**

| Task | Cmd.exe command |
|---|---|
| Copy a file to a different location | `Copy` |
| Change directories | `Cd` |
| Move a file to a new location | `Move` |
| Rename an existing file | `Ren` |

**Table 2.2 Old-style commands for file and folder management *(continued)***

| Task | Cmd.exe command |
|---|---|
| Create a new directory | `MkDir` |
| Remove an empty directory | `RmDir` |
| Delete a file | `Del` |
| Display the contents of a text file | `Type` |

If you have some Unix or Linux shell experience, you may have come up with some alternatives, such as `Ls` for `Dir`, `Cp` for `Copy`, `Rm` instead of `Del`, or `Cat` rather than `Type`. Those are fine answers, and you'll find that they all work within PowerShell, too.

At least, they mostly work. If you tried using extra parameters with some of these commands, you will have realized that these aren't quite the same commands that you're used to. For example, if you tried to run `Dir /s` to get a listing of files and folders, including subdirectories, then you probably got an error message. That's okay—it turns out that this isn't exactly the same `Dir` command you know and love, but it has the same capabilities. We'll cover that a little later in this chapter.

> **TRY IT NOW** Try running `dir /s` in Windows PowerShell just to confirm that it doesn't work. Don't take my word for it!

Have you ever piped a long directory listing to `more`, in order to get a paged result? That same trick still works: `Dir | More`.

> **TRY IT NOW** Change into a long directory, like \Windows\System32, and try running `Dir | More`. You can press Ctrl-C within PowerShell to stop the command from running once you've had enough.

A lot of administrators use the angle bracket to perform redirection. For example, `dir > file.txt` will redirect the output of the `Dir` command into the text file File.txt. You can use the same technique in PowerShell. You can even do the double-angle trick, where the content will be appended to the specified file: `Dir >> file.txt`.

> **TRY IT NOW** Go on—see if you can get a directory listing into a file using this technique, and then append a second directory listing to the end of the same file.

That's the big part of this chapter: you can run commands, and many of the same commands that you've used in Cmd.exe exist, although they may work a bit differently. That leaves us with most of an hour to kill, so let's dig a little deeper.

## 2.4 Accuracy counts

PowerShell is incredibly picky about how you type commands. Command names never contain spaces; it's always `Dir` and never `Di r`. You must have a space after the command name, so that PowerShell knows that the command name is done, and that whatever comes next is a parameter or value.

Technically, `Cd..` is incorrect because it doesn't include a space, and `Cd ..` is correct. In reality, PowerShell v2 catches the `Cd..` error and will do the right thing (move up one level in the directory hierarchy) because that's such a commonly typed command, but that's the only exception that PowerShell will catch that way for you. It won't catch something like `Dir..` so it pays to be careful with those spaces.

`Dir >> file.txt` will redirect a directory to a file; `Dir>>file.txt` will generate an error because the shell will think you've typed a single command name, not two commands connected by angle brackets.

I can't stress enough how important it is to become a neat, careful typist when you're using PowerShell.

## 2.5 Not just files and folders: introducing PSDrives

You know what has always bugged me about Windows? I've spent years memorizing all of these non-intuitive commands, like `Dir` and `Cd`, and they're only good in one place: the filesystem.

The filesystem is a hierarchical database—you probably don't think of the filesystem as a database, but it definitely is. Windows contains lots of other hierarchical databases—the registry comes to mind, as does Active Directory, and there are others—so why can't I use the same commands to manage those databases? Why can't I do any of these:

1. Run `cd hkcu:` to change to the `HKEY_CURRENT_USER` registry hive.
2. Run `dir` to get a list of keys in that hive.
3. Run `cd software` to change to the Software key.
4. Run `dir` to get a list of subkeys.

It turns out that in PowerShell you can do exactly that.

> **TRY IT NOW** Go ahead and try it—run those commands in that order and see if they work for you.

It works because of a PowerShell feature called *PSDrives* (folks usually pronounce that as "Pee-Ess Drives," but it stands for PowerShell Drives). A PSDrive is a mapping between the shell and some kind of data store—the filesystem, the registry, or even Active Directory. As shown in figure 2.2, a *PSDrive provider* sits between the shell and that storage system, making the storage system appear to be a disk drive within the shell.

PSDrive providers can be added into the shell, so that the shell can learn to see other forms of storage. For example, if you install the SQL Server 2008 administrative tools on your computer, you'll gain the ability to map a `SQL:` drive to SQL Server databases. It's pretty cool, and you can use most of the same commands—`Dir`, `Cd`, and so forth—within any PSDrive that you map.

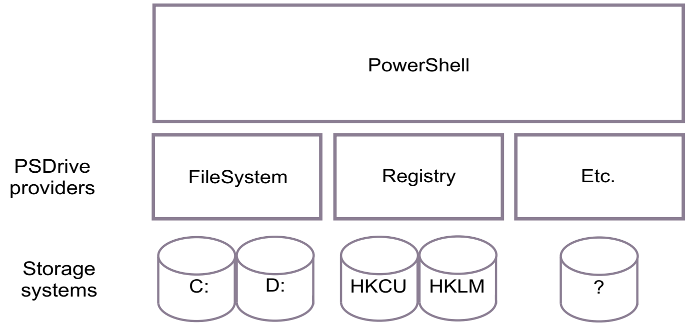



**Figure 2.2 PSDrive providers adapt different forms of storage so they look like disk drives within PowerShell**

There are a few fun facts about PSDrives that you should keep in mind:

- The shell always starts with the same PSDrives mapped. You can run the command `Get-PSDrive` to see them. You'll see one for the `HKEY_CURRENT_USER` (HKCU) and `HKEY_LOCAL_MACHINE` (HKLM) registry hives, one for each local disk, one for environment variables, and one each for PowerShell's function, variable, and alias storage (which we're not going to talk about right now).
- You can map new drives by using the `New-PSDrive` command. Don't bother doing so now, because it's something you'll practice a bit later. Keep in mind that these are PowerShell drives, so you won't see them in Explorer. They only exist within the shell, and whatever you map will unmap automatically when you close the shell. You'll learn how to overcome that shortcoming near the end of the book.
- Unlike the old MS-DOS–style drive names that were limited to a single letter, PSDrives can have longer names, such as `HKCU:` and `HKLM:`. So when you map drives, take the opportunity to make their names more meaningful, like `DEMO:` or `USER:` or `FILES:` rather than `X:`, `Y:`, and `Z:`.
- If you decide to map a new drive using `New-PSDrive`, you'll have to specify a name for the drive (without the colon—it'll just be DEMO or USER or FILES or whatever), the PSDrive provider that will handle the mapping (such as FileSystem), and the source for the mapping (which might be a UNC). For example,

```
New-PSDrive -name DEMO -psprovider FileSystem -root
\\Server\Share\Folder
```

**TRY IT NOW** One thing I found confusing at first was when I was supposed to add the colon and when I shouldn't. Try running `cd hklm` and see what happens; then run `cd hklm:` and see the difference. Whenever you're referring to a drive as part of an action—like changing to it—you'll add the colon to the end of the drive's name.

Spend a few minutes familiarizing yourself with the various default PSDrives. Remember, you can switch to any one of them by using `Cd` and the drive name, such as `Cd Env:` or `Cd C:`. Make sure you can get a directory listing in a variety of drives, and spend a few minutes poking around the Variable: and Env: drives to see what information you find.

PSDrives demonstrate an important design concept behind PowerShell itself: it enables you to leverage existing skills in as many places as possible. For example, rather than learning a whole new set of commands for manipulating the registry, you can use the same commands that you already know from the filesystem. Leveraging existing skills makes you more productive and more effective with less of a learning curve.

## 2.6 Support for external commands

So far, all of the commands you've run in the shell (at least, the ones I've suggested that you run) have been built-in commands, which Windows PowerShell calls *cmdlets* (pronounced "command-lets"). More than 200 of those cmdlets come built into PowerShell, and you can add more—products like Exchange Server, SharePoint Server, and SQL Server all come with add-ins that each include hundreds of additional cmdlets.

But you're not limited to the cmdlets that come with PowerShell—you can also use the same external command-line utilities that you have probably been using for years, including Ping, Nslookup, Ipconfig, Net, and so forth. Because these aren't native PowerShell cmdlets, you use them the same way that you always have. PowerShell will launch Cmd.exe behind the scenes, because it knows how to run those external commands, and any results will be displayed within the PowerShell window. Go ahead and try a few old favorites right now. For example, I'm often asked how you can use PowerShell to map a regular network drive—one that can be seen from within Explorer. I always use `Net Use`, myself, and it works fine within PowerShell.

**TRY IT NOW** Try running some external command-line utilities that you've used before. Do they work the same? Do any of them fail?

The `Net Use` example illustrates a really important lesson: with PowerShell, Microsoft (perhaps for the first time ever) isn't saying, "you have to start over and learn everything from scratch." Instead, Microsoft is saying, "if you already know how to do something, keep doing it that way. We'll try to provide you with better and more complete tools going forward, but what you already know will still work." One reason there's no "Map-Drive" command within PowerShell is that `Net Use` already does a good job, so why not keep using it?

There are certainly instances where Microsoft has provided better tools than some of the existing, older ones. For example, the native `Test-Connection` cmdlet provides more options and more flexible output than the old, external `Ping` command—but if you know how to use `Ping`, and it's meeting whatever need you have, then go right on using it. It will work fine from within PowerShell.

All that said, I do have to deliver a harsh truth: not every single external command will work flawlessly from within PowerShell, at least not without a little tweaking on your part. That's because PowerShell's parser—the bit of the shell that reads what you've typed and tries to figure out what you want the shell to do—doesn't always guess correctly. Sometimes, you'll type an external command and PowerShell will mess up, start spitting out errors, and just generally not work.

For example, things can get tricky when an external command has a lot of parameters—that's where I see PowerShell break the most. We're not going to dive into the details of why it works, but here's a way to run a command that will ensure its parameters work properly:

```
$exe = "C:\Vmware\vcbMounter.exe"
$host = "server"
$user = "joe"
$password = "password"
$machine = "somepc"
$location = "somelocation"
$backupType = "incremental"

& $exe -h $host -u $user -p $password -s "name:$machine" -r $location -t
$backupType
```

This supposes that you have an external command named vcbMounter.exe (which is a real-life command-line utility supplied with some of VMWare's virtualization products). It accepts several parameters:

- `-h` for the host name
- `-u` for the user name
- `-p` for the password
- `-s` for the server name
- `-r` for a location
- `-t` for a backup type

What I've done is put all the various elements—the executable path and name, as well as all of the parameter values—into placeholders, which start with the `$` character. That forces PowerShell to treat those values as single units, rather than trying to parse them to see if any of them contain commands or special characters or anything. Then I used the invocation operator, passing it the executable name, all of the parameters, and the parameters' values. That pattern will work for almost any command-line utility that's being grumpy about running within PowerShell.

## 2.7 The same old commands—almost

Let's put external commands on the back burner for a moment and get back to the native commands. After all, those are the really interesting ones because they're the ones that make PowerShell more than just a copy of Cmd.exe.

At the beginning of this chapter, you saw how commands like `Dir`, `Cd`, `Type`, and so forth all worked within PowerShell. You also saw how they didn't necessarily work exactly the same—running `Dir /s`, for example, causes an error. Why is that?

The truth is that PowerShell doesn't actually contain a `Dir` command, or a `Type` command, or any of those other commands. Instead, PowerShell defines those as *aliases* to some of PowerShell's native cmdlets. Aliases are just nicknames for cmdlet names. These are some of the real cmdlet names you've been using:

- `Get-ChildItem` (for `Dir`, `Ls`)
- `Set-Location` (for `Cd`)
- `Move-Item` (for `Move`)
- `Rename-Item` (for `Ren`)
- `Remove-Item` (for `Del`, `Rm`, `RmDir`)
- `Copy-Item` (for `Copy`, `Cp`)
- `Get-Content` (for `Type`, `Cat`)
- `New-Item` (for `MkDir`)

Those cmdlet names are obviously longer, making them harder to type, so Microsoft added those aliases as a way of saving your fingers some wear and tear. Also, by selecting aliases that match the old Cmd.exe-style names (as well as Linux and Unix names), the company gave you a way of jumping right into PowerShell and performing basic tasks without having to spend too much up-front time learning new command names.

That explains why `Dir /s` doesn't work: you're not running the `Dir` command from your past, and `Get-ChildItem` doesn't support a `/s` parameter. `Get-ChildItem` can do the same thing as `Dir /s`, but you'll have to learn a new parameter name, which is `-recurse`.

In fact, this is probably a good time to point out some common characteristics about PowerShell cmdlets:

- All PowerShell cmdlet names have a strict naming convention. Cmdlet names start with a verb, like Get or Copy, followed by a hyphen, and then a singular noun, such as Item or Content. The list of allowed verbs is quite small—a few dozen or so—and the number of verbs you use on a daily basis will probably number less than a dozen. The idea is that you'll gradually become used to those verbs and be able to guess new cmdlet names. More on that in a second.
- Cmdlet names tend to be a little generic. Why `Move-Item` and not Move-File? Keep in mind that the cmdlet has to operate in the registry, environment variables, and other storage systems, as well as the filesystem. Rather than having separate Move-File and Move-RegistryKey cmdlets, PowerShell has a single generic `Move-Item`.
- Parameter names (`-recurse` was one example) always start with a dash, and for parameters that accept a value (like the `-name DEMO` example I showed you earlier), there's always a space separating the parameter name and the value. Dash, name, space, value. When I teach classes, I make my students repeat that aloud:

dash, name, space, value. After that, they never wonder if parameter names should start with a dash or a slash, or if there's supposed to be an equal sign or colon in between the name and value. It's "dash, name, space, value," and never anything else.

- Parameter names are used consistently throughout the shell. If one cmdlet has a `-computerName` parameter, which is used for specifying a computer name, then most cmdlets that need a computer name will also have a `-computerName` parameter.
- Both parameter and cmdlet names are intended to be clear and meaningful. When you look at a cmdlet name like `Get-Content`, you should be pretty clear that it's getting some kind of content from something. A parameter name like `-credential` doesn't leave much to the imagination, either—you should be pretty certain what that parameter is going to do.
- Because "clear" can sometimes mean "lengthy," Microsoft gives you shortcuts for cmdlet names and parameter names, to save you some typing. Cmdlet names can be given shorter aliases, as we've already discussed, and parameter names don't need to be typed in their entirety. For example, if `-recurse` is the only parameter of `Get-ChildItem` that starts with the letter *r*, then you only have to type `-r` and PowerShell will know what you mean. If a cmdlet has both a `-computerName` and `-credential` parameter, typing `-comp` will probably be enough for PowerShell to figure out that you mean the `-computerName` parameter.

This is another good place for me to remind you to be a neat, careful typist. `Get-ChildItem-recurse` is incorrect, because there's no space between the end of the cmdlet name and the dash that starts the parameter name. `Get-ChildItem -recurse` is correct, with that space in between the two elements. It's very, very important that you focus on those little typing details, because getting them wrong will generate sometimes-confusing errors that will do nothing but slow you down.

PowerShell is all about consistency. That's not to say it's 100 percent consistent, because it is, after all, the product of many human beings, who do tend to make mistakes sometimes. But it's pretty consistent most of the time. In the previous chapter, I wrote about how graphical user interfaces (GUIs) offer features to help you figure out what you can do with them: right-click menus, tooltips, menus, and so forth. Programmers refer to those features as *discoverability features*, because they literally help you discover the tool's capabilities. A command-line interface (CLI) like PowerShell lacks those kinds of graphical discoverability features, but consistency can provide a kind of discoverability of its own.

For example, let's say I told you that the cmdlet verb Get was used for all cmdlets that retrieve or display something (as with `Get-Content`). You already know that cmdlet nouns are singular, and never plural. Can you guess the names of the cmdlets that would perform the tasks in table 2.3?

**Table 2.3 Guessing the cmdlet names for specific tasks**

| Task | Cmdlet |
|---|---|
| Display a list of services | |
| Display a list of running processes | |
| Display the contents of an event log | |
| Create a new service | |
| Retrieve Exchange mailboxes | |
| Create a new Exchange mailbox | |

All of those tasks can be accomplished with one of two verbs: Get or New. From there, you just have to make an educated guess about the noun. The right answers are shown in table 2.4.

**Table 2.4 Introducing some new cmdlets**

| Task | Cmdlet |
|---|---|
| Display a list of services | `Get-Service` |
| Display a list of running processes | `Get-Process` |
| Display the contents of an event log | `Get-EventLog` |
| Create a new service | `New-Service` |
| Retrieve Exchange mailboxes | `Get-Mailbox` |
| Create a new Exchange mailbox | `New-Mailbox` |

The last two cmdlets in table 2.4 aren't native to PowerShell, and you won't be able to try them unless you have the Exchange Server 2007 (or 2010) add-in loaded, which isn't something we're going to cover just now. The point for right now is guessing the right cmdlet name—if you were able to do that, then you're well on your way to mastering the shell.

More importantly, don't ever be afraid to guess a cmdlet name, and don't be afraid to be wrong. In the next chapter, I'll show you how to check and see if your guesses are correct, and how to teach yourself how to use a cmdlet once you've discovered its name. I'll also show you how to search for cmdlets based on a part of their name, which can be another useful trick for discovering new cmdlets.

## 2.8 Common points of confusion

Whenever it seems appropriate, I'll wrap up each chapter with a brief section that covers some of the common mistakes I see when I teach classes. The idea is to help you see what most often confuses other administrators like yourself, and to avoid those problems—or at least to be able to find a solution for them—as you start working with the shell.

### 2.8.1 Typing cmdlet names

First up is the typing of cmdlet names. It's always Verb-Noun, like `Get-Content`. All of these are things I see newcomers try, but they won't work:

- Get Content
- GetContent
- Get=Content
- Get_Content

### 2.8.2 Typing parameters

Parameters are also consistently written. A parameter that takes no value, such as `-recurse`, just gets a dash before its name. There need to be spaces separating the cmdlet name from its parameters, and the parameters from each other. These are all correct:

- `Dir -rec` (the shortened parameter name is fine)
- `New-PSDrive -name DEMO -psprovider FileSystem -root \\Server\Share`

But these examples are all incorrect:

- `Dir-rec` (no space between alias and parameter)
- `New-PSDrive -nameDEMO` (no space between parameter name and value)
- `New-PSDrive -name DEMO-psprovider FileSystem` (no space between the first parameter's value and the second parameter's name)

PowerShell isn't normally picky about upper- and lowercase, meaning that `dir` and `DIR` are the same, as are `-RECURSE` and `-recurse` and `-Recurse`. But the shell sure is picky about those spaces and dashes!

## 2.9 Lab

Because this is the book's first lab, I'll take a moment and describe how these are supposed to work. For each lab, I'll give you a few tasks that you can try and complete on your own. Sometimes I'll provide a hint or two to get you going in the right direction. From there, you're on your own.

I absolutely guarantee that everything you need to know to complete every lab is either in that same chapter or was covered in a previous chapter (and the "previously covered" info is the stuff I'm most likely going to give you a hint for). I'm not saying the answer is going to be right out in plain sight: most often, a chapter will have taught you how to discover something on your own, and you'll have to go through that discovery process to find the answer. It might seem frustrating, but forcing yourself to do it will absolutely make you more successful with PowerShell in the long run. I promise.

Keep in mind that you can find sample answers at MoreLunches.com. My answers might not exactly match yours, and that will become increasingly true as we move on to more complex material. In fact, you'll often find that PowerShell offers a half-dozen ways to accomplish almost anything. I'll show you the way I use the most, but if you come up with something different, you're not wrong! Any way that gets the job done is correct.

Using just what you learned in this chapter, complete the following tasks in Windows PowerShell:

1. Create a text file that contains the names of the files and folders in C:\Windows (don't worry about including subdirectories—that would take too long). Name the text file MyDir.txt.
2. Display the contents of that text file.
3. Rename the file from MyDir.txt to WindowsDir.txt.
4. Create a new folder named LabOutput—you can either do this in your Documents folder, or in the root of your C: drive.
5. Copy WindowsDir.txt into the LabOutput folder.
6. Delete the original copy of WindowsDir.txt—not the copy that you just made in LabOutput.
7. Display a list of running processes.
8. Redirect a list of running processes into a file named Procs.txt.
9. Move Procs.txt into the LabOutput folder if it isn't in there already.
10. Display the contents of Procs.txt so that only one page displays at a time (remember the trick with `| more`).

Hopefully these tasks seem straightforward for you. If so—excellent! You were leveraging your existing command-line skills to make PowerShell perform a few practical tasks for you. If you're new to the command-line world, these tasks are a good introduction to what you'll be doing in the rest of this book.

I'm not going to hit you with any "Ideas for on your own" in this chapter. Because we're just beginning, I'll be happy with the tasks you've already completed. If you didn't get a chance to try all of the "Try it Now" examples in this chapter, go back and do so now if you have time, and make sure that you're able to accomplish all ten of the preceding lab tasks.

# 3 Using the help system

In the first chapter of this book, I mentioned that discoverability is a key feature that makes graphical user interfaces (GUIs) easier to learn and use, and that command-line interfaces (CLIs) like PowerShell are often more difficult because they lack those discoverability features. In fact, PowerShell has fantastic discoverability features—they're just not that obvious. One of the main discoverability features is the help system.

## 3.1 The help system: how you discover commands

Bear with me for a minute while I climb up on a soapbox and preach to you.

We work in an industry that doesn't place a lot of emphasis on reading, although we do have an acronym, *RTFM*, that we cleverly pass along to users when we wish *they* would "read the friendly manual." Most administrators tend to dive right in, relying on things like tooltips, context menus, and so forth—those GUI discoverability tools—to figure out how to do something. I know that's how I often work, and I imagine you do the same thing. Let me be clear about one thing:

> *If you aren't willing to read PowerShell's help files, you won't be effective with PowerShell. You won't learn how to use it, you won't learn how to administer products like Windows and Exchange with it, and you might as well stick with the GUI.*

That's about as clear as I can be. It's a very blunt statement, I know, but it's absolutely true. Imagine trying to figure out Active Directory Users and Computers, or any other administrative console, without the help of tooltips, menus, and context menus! Trying to learn and use PowerShell without taking the time to read and understand the help files is the same thing. It'll be frustrating, confusing, and ineffective. Why?

- If you need to perform a task and don't know what command to use, the help system is how you'll find that command. Not Google or Bing, but the help system.
- If you run a command and get an error, the help system is what will show you how to properly run the command so that you don't get errors.
- If you want to link multiple commands together to perform some complex task, the help system is what will show you how each command is able to connect to others. You don't need to search for examples on Google or Bing; you need to learn how to use the commands themselves, so that you can create your own examples and solutions.

I know, this preaching of mine is a little heavy-handed. The problem is that 90 percent of the problems I see students struggling with in classes, and on the job, could be solved if those folks took a few minutes to sit back, breathe deeply, and read the help. Of course, you need to understand what you're reading, and that's what this chapter is all about.

From here on out, this book is going to do a couple of things to help encourage you to read the help:

- Although I will be showing you many commands in my examples, I will almost never expose the complete functionality, options, and capabilities of each command. You should read the help for each and every command I show you, so that you'll be familiar with the additional things that command can do.
- In the labs, I may give you a hint about which command to use for a task, but I won't give you hints about the syntax. You'll need to use the help system to discover that syntax on your own in order to complete the labs.
- I'll often ask you to identify new commands or parameters as part of the labs and "Ideas for on your own" sections. The ideas is for you to practice using the help system itself, because the more proficient you are with the help system, the more proficient you'll be with the shell.

I absolutely promise you that mastering the help system is the secret recipe for becoming a PowerShell expert. No, you won't find every little detail in there, and there's a lot of super-advanced stuff that isn't documented in the help system, but in terms of being an effective day-to-day administrator, the help system is the key. This book will make that system understandable, and it will teach you the concepts that the help skips over, but it will only do so in conjunction with the built-in help.

Stepping off my soapbox now.

## 3.2 Asking for help

Windows PowerShell provides a cmdlet, `Get-Help`, that accesses the help system. You may see examples of people using the `Help` keyword instead, or even the `Man` keyword (which comes from Unix and means "Manual"). `Man` and `Help` aren't aliases at all—they are *functions*, which are basically wrappers around the core `Get-Help` cmdlet. `Help` works much like the base `Get-Help`, but it pipes the help output to `More` so that you get a nice paged view instead of seeing all the help fly by at once. Running `Help Get-Content` and `Get-Help Get-Content` produces the same results, but the first one has a page-at-a-time display. You could run `Get-Help Get-Content | More` to produce that same paged display, but it's a lot more typing. I'll typically just use `Help`, but I want you to understand that there's some trickery going on under the hood.

By the way, sometimes that paginated display gets annoying—you've got the information you need, but it still wants you to hit the spacebar to display the remaining information. If that's ever the case, press Ctrl-C to cancel the command and return to the shell prompt. Within the shell's console window, Ctrl-C always means "break" rather than "copy to the clipboard." In the more graphically oriented Windows PowerShell ISE, however, Ctrl-C does in fact copy to the clipboard. There's a red "stop" button in the toolbar that will stop a running command.

The help system has two main goals: to help you find commands to perform specific tasks, and to help you learn how to use those commands once you've found them.

## 3.3 Using help to find commands

Technically speaking, the help system has no idea what commands are present in the shell. All it knows is what help topics are available. Fortunately, Microsoft ships a help topic for nearly every cmdlet that they produce, so there's usually no difference. In addition, the help system can also access information that isn't related to a specific cmdlet, including background concepts and other general information.

Like most commands, `Get-Help` (and therefore `Help`) has several parameters. One of those—perhaps the most important one—is `-Name`. It's a positional parameter, so you don't have to type `-Name` and can simply provide the name you're looking for. This parameter specifies the name of the help topic you'd like to access, and it accepts wildcards. This ability to handle wildcards is what makes the help system useful for discovering commands.

For example, suppose I want to do something with an event log. I don't know what commands might be available, so I want to search and see what help topics talk about event logs. I might run either of these two commands:

```
Help *log*
Help *event*
```

The first of those commands returns a list like this on my computer:

```
Name
----
Get-EventLog
Clear-EventLog
Write-EventLog
Limit-EventLog
Show-EventLog
New-EventLog
Remove-EventLog
```

```
about_eventlogs
about_logical_operators
```

Most of those seem to have something to do with event logs, and based on the Verb-Noun naming format, all but the last two appear to be help topics related to specific cmdlets. The last two "about" topics provide background information. The last one doesn't seem to have anything to do with event logs, but it came up because it does have "log" in it—part of the word "logical." Whenever possible, I try to search using the broadest term possible—"*event*" or "*log*" as opposed to "*eventlog*"—because I'll get the most results possible.

Once you have a cmdlet that you think will do the job—`Get-EventLog` looks like a good candidate for what I'm after right now—you can ask for help on that specific help topic:

```
Help Get-EventLog
```

Here's another cool trick that PowerShell offers: tab completion. This enables you to type a portion of a command name, and then press Tab. The shell will complete what you've typed with the closest match; you can continue pressing Tab to cycle through alternative matches.

> **TRY IT NOW** Type `Help Get-Ev` and press Tab. My first match is `Get-Event`, which isn't what I want; pressing Tab again brings up `Get-EventLog`, which is what I'm after. I can hit Return to accept the command and display the help for that cmdlet.

You can continue to use wildcards. If PowerShell only finds one match to whatever you've typed, it won't display a list of topics with just that one item. Instead, it will display the contents for that item.

> **TRY IT NOW** Run `Help Get-EventL*` and you should see the help file for `Get-EventLog`, rather than a list of matching help topics.

If you've been following along in the shell, you should now be looking at the help file for `Get-EventLog`. This is called the *summary help*, and it's meant to be a short description of the command and a reminder of the syntax. This is useful when you need to quickly refresh your memory on a command's usage, and it's where we'll begin interpreting the help file itself.

### Above and beyond

Sometimes, I'll need to share a little bit of information that, although nice, isn't essential to your understanding of the shell. I'll put that information into an "Above and beyond" section, like this one. If you skip these, you'll be fine; if you read them, you'll often learn about an alternative way to do something, or get a bit of additional insight into PowerShell.

***(continued)***

I mentioned that the `Help` command doesn't actually search for cmdlets; it searches for help topics. When every cmdlet has a help file, that works out to pretty much the same thing. But there is a way to directly search for cmdlets: the `Get-Command` cmdlet. It has an alias, `Gcm`, which makes typing it a bit easier.

Like the `Help` cmdlet, `Gcm` accepts wildcards, meaning that you can run something like `Gcm *event*` to see all commands that contain "event" in their name. For better or worse, that list will include not only cmdlets, but also external commands like netevent.dll, which may not be very useful.

A better approach is to use the `-Noun` or `-Verb` parameters. Because only cmdlet names have nouns and verbs, the results will be limited to cmdlets. `Gcm -noun *event*` will return a list of cmdlets dealing with events; `Gcm -verb Get` will return all cmdlets capable of retrieving things. You can also use the `-CommandType` parameter, specifying a type of cmdlet: `Gcm *log* -type cmdlet` will show a list of all cmdlets that include "log" in their names, and the list won't include any external applications or commands.

## 3.4 Interpreting the help

PowerShell's cmdlet help files have a particular set of conventions. Learning to understand what you're looking at is the key to extracting the maximum amount of information from these files, and to learning to use the cmdlets themselves more effectively.

### 3.4.1 Parameter sets and common parameters

Most commands can work in a variety of different ways, depending on what you need them to do. For example, here's the syntax section for the `Get-EventLog` help:

```
SYNTAX
    Get-EventLog [-AsString] [-ComputerName <string[]>] [-List] [<Com
    monParameters>]

    Get-EventLog [-LogName] <string> [[-InstanceId] <Int64[]>] [-Afte
    r <DateTime>] [-AsBaseObject] [-Before <DateTime>] [-ComputerName
     <string[]>] [-EntryType <string[]>] [-Index <Int32[]>] [-Message
     <string>] [-Newest <int>] [-Source <string[]>] [-UserName <strin
    g[]>] [<CommonParameters>]
```

Notice that the command is listed twice. That indicates that the command supports two *parameter sets*, there are two distinct ways in which you can use this command. Some of the parameters will be shared between the two sets. You'll notice, for example, that both parameter sets include a `-ComputerName` parameter. But the two parameter sets will always have at least one unique parameter that exists only in that parameter set. In this case, the first set supports `-AsString` and `-List`, neither of which are included in the second set; the second set contains numerous parameters that aren't included in the first.

Here's how this works: if you use a parameter that's only included in one set, you're locked into that set and can only use additional parameters that appear within that same set. If I choose to use `-List`, then the only other parameters I can use are `-AsString` and `-ComputerName`, because those are the only other parameters included in the parameter set where `-List` lives. I couldn't add in the `-LogName` parameter, because it doesn't live in the first parameter set. That means `-List` and `-LogName` are *mutually exclusive*—you'll never use both of them at the same time, because they live in different parameter sets.

Sometimes it's possible to run a command with only parameters that are shared between multiple sets. In those cases, the shell will usually select the first-listed parameter set. Because each parameter set implies different behavior, it's important to understand which parameter set you're running.

You'll notice that every parameter set for every PowerShell cmdlet ends with `[<CommonParameters>]`. This refers to a set of eight parameters that are available on every single cmdlet, no matter how you're using that cmdlet. We're not going to discuss those common parameters now, but we'll discuss some of them later in this book, when we get to using them for a real task. Later in this chapter, though, I'll show you where to learn more about those common parameters, if you're interested.

### 3.4.2 Optional and mandatory parameters

Not every single parameter is needed in order to make a cmdlet run. PowerShell's help lists optional parameters in square brackets. For example, `[-ComputerName <string[]>]` indicates that the entire `-ComputerName` parameter is optional. You don't have to use it at all—the cmdlet will probably default to the local computer if you don't specify an alternative name using this parameter. That's also why `[<CommonParameters>]` is in square brackets—you can run the command without using any of the common parameters.

Almost every cmdlet has at least one optional parameter. You may never need to use some of these parameters, and others may be used on a daily basis. Keep in mind that, when you choose to use a parameter, you only have to type enough of the parameter name so that PowerShell can unambiguously figure out which parameter you meant. `-L` wouldn't be sufficient for `-List`, for example, because `-L` could also mean `-LogName`. But `-Li` would be a legal abbreviation for `-List`, because no other parameter starts with `-Li`.

What if you try to run a command and forget one of the mandatory parameters? Take a look at the help for `Get-EventLog`, for example, and you'll see that the `-LogName` parameter is mandatory—the parameter isn't enclosed in square brackets. Try running `Get-EventLog` without specifying a log name.

> **TRY IT NOW** Follow along on this example—run `Get-EventLog` without any parameters.

PowerShell should have prompted you for the mandatory `LogName` parameter. If you type something like `System` or `Application` and hit Return, the command will run correctly. You could also press Ctrl-C to abort the command.

### 3.4.3 Positional parameters

PowerShell's designers knew that some parameters would be used so frequently that you wouldn't want to continually type the parameter name. Those commonly used parameters are often *positional*, meaning that you can provide a value without typing the parameter's name, provided you put that value in the correct position.

There are two ways to identify a positional parameter. The first way is right in the syntax summary: the parameter name—just the name—will be surrounded by those square brackets. For example, look at the first two parameters in the second parameter set of `Get-EventLog`:

```
[-LogName] <string> [[-InstanceId] <Int64[]>]
```

The first parameter, `-LogName`, isn't optional. I can tell because the entire parameter—its name and its value—aren't surrounded by square brackets. The parameter name, however, is enclosed in square brackets, so that's a positional parameter. I could provide the log name without having to type `-LogName`. Because this appears in the first position within the help file, I know that the log name is the first parameter I have to provide.

The second parameter, `-InstanceId`, is optional—both it and its value are enclosed in square brackets. Within those, `-InstanceId` itself is also contained in square brackets, indicating that this is also a positional parameter. It appears in the second position, so I would need to provide a value in the second position if I chose to omit the parameter name.

The `-Before` parameter is optional, because it's entirely enclosed within square brackets. The `-Before` name isn't in square brackets, which tells me that if I choose to use that parameter, I must type the parameter name (or at least a portion of it).

There are some tricks to using positional parameters:

- It's okay to mix and match positional parameters with those that require their names. Positional parameters must always be in the correct position. For example, `Get-EventLog System -Newest 20` is legal. `System` will be fed to the `-LogName` parameter, because that value is in the first position; `20` will go with the `-Newest` parameter because the parameter name was used.
- It's always legal to specify parameter names, and when you do so, the order in which you type them isn't important. `Get-EventLog -newest 20 -Log Application` is legal because I've used parameter names (in the case of `-LogName`, I abbreviated it).
- If you use multiple positional parameters, don't lose track of their positions. `Get-EventLog Application 0` will work, with `Application` being attached to

`-LogName` and `0` being attached to `-InstanceId`. `Get-EventLog 0 Application` won't work, because `0` will be attached to `-LogName`, and there is no log named `0`.

I'll offer a best practice: use parameter names until you become comfortable with a particular cmdlet and get tired of typing a commonly used parameter name over and over. After that, use positional parameters to save yourself typing. When the time comes to paste a command into a text file for easier reuse, always use the full cmdlet name and type out the complete parameter name—no positional parameters and no abbreviated parameter names. Doing so makes that file easier to read and understand in the future, and because you won't have to type the parameter names (that's why you pasted the command into a file, after all), you won't be creating extra typing work for yourself.

I said there were two ways to locate positional parameters. The second requires that you open the help file using the `-full` parameter of the `Help` command.

**TRY IT NOW** Run `Help Get-EventLog -full`. Remember to use the spacebar to view the help file one page at a time, and to press Ctrl-C if you want to stop viewing the file before reaching the end. For now, page through the entire file, so that you can scroll back and review it all.

Page down until you see the help entry for the `-LogName` parameter. It should look something like this:

```
-LogName <string>
    Specifies the event log.  Enter the log name (the value of th
    e Log property; not the LogDisplayName) of one event log. Wil
    dcard characters are not permitted. This parameter is require
    d.

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

Here, I can see that this is a mandatory parameter—it's listed as required. Further, it's a positional parameter, and it occurs in the first position, right after the cmdlet name.

I always encourage students to focus on reading this full help, rather than the abbreviated syntax reminder when they're getting started with a cmdlet. Doing so reveals more details, including that description of what the parameter is used for. I can also see that this parameter doesn't accept wildcards, which means I can't provide a value like `App*`—I need to type out the full log name, such as `Application`.

### 3.4.4 Parameter values

The help files also give you clues about what kind of input each parameter accepts. Some parameters, referred to as *switches*, don't require any input value at all. In the abbreviated syntax, they look like this:

```
[-AsString]
```

And in the full syntax, they look like this:

```
-AsString [<SwitchParameter>]
    Returns the output as strings, instead of objects.

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

The `[<SwitchParameter>]` part confirms that this is a switch, and that it doesn't expect an input value. Switches are never positional; you always have to type the parameter name (or at least an abbreviated version of it). Switches are always optional, so that you have the choice to use them or not.

Other parameters expect some kind of input value, which will always follow the parameter name and be separated from the parameter name by a space (and not by a colon, equal sign, or any other character). In the abbreviated syntax, the type of input expected is shown in angle brackets, like `< >`:

```
[-LogName] <string>
```

It's shown the same way in the full syntax:

```
-Message <string>
    Gets events that have the specified string in their messages.
     You can use this property to search for messages that contai
    n certain words or phrases. Wildcards are permitted.

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

These are some common types of input:

- `String`—A series of letters and numbers. These can sometimes include spaces, but when they do, the entire string must be contained within quotation marks. For example, a string value like `C:\Windows` doesn't need to be enclosed in quotes, but `C:\Program Files` does, because it has that space in the middle. For now, you can use single or double quotation marks interchangeably, but it's best to stick with single quotes.
- `Int`, `Int32`, *or* `Int64`—An integer number (a whole number with no decimal portion).
- `DateTime`—Generally, a string that can be interpreted as a date based on your computer's regional settings. In the U.S., that's usually something like `10-10-2010`, with the month, day, and year.

There are other, more specialized types, and we'll discuss those as we come to them.

You'll also notice some values that have more square brackets:

```
[-ComputerName <string[]>]
```

The side-by-side brackets after `string` don't indicate that something is optional. Instead, `string[]` indicates that the parameter can accept an *array*, or *collection*, or *list* of strings. In these cases, it's always legal to provide a single value:

```
Get-EventLog Security -computer Server-R2
```

But it's also legal to specify multiple values. A simple way to do so is to provide a comma-separated list. PowerShell treats all comma-separated lists as arrays of values:

```
Get-EventLog Security -computer Server-R2,DC4,Files02
```

Once again, any individual value that contains a space must be enclosed in quotation marks. However, the entire list doesn't get enclosed in quotation marks—it's important that only individual values be in quotes. The following is legal:

```
Get-EventLog Security -computer 'Server-R2','Files02'
```

Even though neither of those values needs to be in quotation marks, it's okay to use the quotes if you want to. But the following is wrong:

```
Get-EventLog Security -computer 'Server-R2,Files01'
```

In this case, the cmdlet will be looking for a single computer named `Server-R2,Files01` which is probably not what you wanted.

Another way to provide a list of values is to type them into a text file, with one value per line. Here's an example:

```
Server-R2
Files02
Files03
DC04
DC03
```

Then, you can use the `Get-Content` cmdlet to read the contents of that file, and send those contents into the `-computerName` parameter. The way to do this is to force the shell to execute the `Get-Content` command first, so that the results get fed to the parameter.

Remember in high school math how parentheses, like `( )`, could be used to specify the order of operations in a mathematical expression? The same thing works in PowerShell: by enclosing a command in parentheses, you force that command to execute first:

```
Get-EventLog Application -computer (Get-Content names.txt)
```

This is a really useful trick. I keep text files with the names of different classes of computers—web servers, domain controllers, database servers, and so forth—and then use this trick to run commands against entire sets of computers.

There are a few other ways to feed a list of values to a parameter, including reading computer names from Active Directory. Those techniques are a bit more complex, though, so we'll come to them in later chapters, after learning some of the cmdlets needed to make the trick work.

There's one more way that you can specify multiple values for a parameter, provided it's a mandatory parameter: don't specify the parameter at all. As with all mandatory parameters, PowerShell will prompt you for the parameter value. For parameters that accept multiple values, you can type the first value and press Return. PowerShell will then prompt for a second parameter, which you can type and finish by hitting Return. Keep doing that until you're finished, and press Return on a blank prompt to let PowerShell know that you're finished. As always, you can press Ctrl-C to abort the command if you don't want to be prompted for entries.

### 3.4.5 Examples

I tend to learn by example, which is why I'll try to squeeze as many examples into this book as possible. PowerShell's designers know that most administrators enjoy having examples, so they built a lot of them into the help files. If you've scrolled to the end of the help file for `Get-EventLog`, you probably noticed almost a dozen examples of how to use the cmdlet.

There's an easier way to get to those examples, if they're all you want to see: use the `-example` parameter of the `Help` command, rather than the `-full` parameter.

```
Help Get-EventLog -example
```

> **TRY IT NOW** Go ahead and pull up the examples for a cmdlet using this new parameter.

I love having these examples, even though some of them can get pretty complicated. If an example looks too complicated for you, just ignore it and examine the others for now. Or, experiment a bit (always on a non-production computer, of course) to see if you can figure out what the example does, and why.

## 3.5 Accessing "about" topics

Earlier in this chapter, I mentioned that PowerShell's help system includes information on background topics, as well as help for specific cmdlets. These background topics are often called "about" topics, because their filenames all start with `about_`. You may also recall from earlier in this chapter that all cmdlets support a set of common parameters. How do you think you could learn more about those common parameters?

> **TRY IT NOW** Before you read ahead, see if you can list the common parameters by using the help system.

I would start by using wildcards. Because the word "common" has been used repeatedly here in the book, that's probably a good keyword to start with:

```
Help *common*
```

It's such a good keyword, in fact, that it will match only one help topic: About_common_parameters. That topic will display automatically because it's the only match. Paging through the file a bit, you'll find a list of the eight common parameters:

```
-Verbose
-Debug
-WarningAction
-WarningVariable
-ErrorAction
-ErrorVariable
-OutVariable
-OutBuffer
```

The file says that there are two additional "risk mitigation" parameters, but those aren't supported by every single cmdlet.

The "about" topics in the help system are tremendously important, but because they're not related to a specific cmdlet, they can be easy to overlook. Try running `help about*` for a list of all of them, and you might be surprised at how much extra documentation is hidden away inside the shell.

There are several third-party scripts and applications that can make PowerShell's help easier to access. At http://mng.bz/5w8E, you'll find a PowerShell script that constructs a graphical browser that lists all of the available help topics. At http://www.primaltools.com/downloads/communitytools/ you'll find a dedicated Windows application that does much the same thing. If you're an iPhone or iPad or iPod touch user, there's an application called iPowerShell that provides handy access to the help files that come with PowerShell v2, Exchange Server, and a few other products. Log on to http://download.microsoft.com and enter "PowerShell Help" as a search term, and you'll find a downloadable Windows Help File that includes the help (including the "about" topics) that comes with PowerShell.

## 3.6 Accessing online help

PowerShell's help files were written by mere human beings, which means they're not, unfortunately, error-free. Updating the help files can be tough, because they're technically part of the operating system. Microsoft won't issue a hotfix for typos, and it's tough to even get that kind of non-critical content into a service pack. What Microsoft can do, however, is update a website.

The `-online` parameter of PowerShell's help command will attempt to open the web-based help for a given command:

```
Help Get-EventLog -online
```

The help is hosted on Microsoft's TechNet website, and it's always going to be more up to date than what's installed with PowerShell itself. So if you think you've spotted an error in an example or in the syntax, try viewing the online version of the help. Not every single cmdlet in the universe has online help; it's up to each product team (like the Exchange team, the SQL Server team, the SharePoint team, and so forth) to provide that help. But when it's available, it's a nice companion to what's built in.

## 3.7 Lab

Hopefully, this chapter has conveyed the importance of mastering the help system in PowerShell. Now it's time to hone your skills by completing the following tasks. Keep in mind that sample answers can be found on MoreLunches.com. Look for *italicized* words in these tasks, and use them as clues to complete that task.

1. Can you find any cmdlets capable of converting other cmdlets' output into *HTML*?
2. Are there any cmdlets that can redirect output into a *file*, or to a *printer*?
3. How many cmdlets are available for working with *processes*? (Hint: Remember that cmdlets all use a singular noun.)
4. What cmdlet might you use to *write* to an event *log*?
5. You've learned that aliases are nicknames for cmdlets; what cmdlets are available to create, modify, export, or import *aliases*?
6. Is there a way to keep a *transcript* of everything you type in the shell, and save that transcript to a text file?
7. It can take a long time to retrieve all of the entries from the Security *event* log. How can you get just the 100 most recent entries?
8. Is there a way to retrieve a list of the *services* that are installed on a remote computer?
9. Is there a way to see what *processes* are running on a remote computer?
10. Examine the help file for the `Out-File` cmdlet. The files created by this cmdlet default to a width of how many characters? Is there a parameter that would enable you to change that width?
11. By default, `Out-File` will overwrite any existing file that has the same filename as what you specify. Is there a parameter that would prevent the cmdlet from overwriting an existing file?
12. How could you see a list of all *aliases* defined in PowerShell?
13. Using both an alias and abbreviated parameter names, what is the shortest command line you could type to retrieve a list of running processes from a computer named Server1?
14. How many cmdlets are available that can deal with generic objects? (Hint: Remember to use a singular noun like "object" rather than a plural one like "objects").
15. This chapter briefly mentioned *arrays*. What help topic could tell you more about them?

16 The `Help` command can also search the contents of a help file. Are there any topics that might explain any *breaking* changes between PowerShell v1 and PowerShell v2?

## 3.8 Ideas for on your own

You're going to be using the help files a lot throughout this book. Personally, I find it frustrating to be in the middle of a command line and then realize I need to look something up in the help, because it means I have to stop what I'm typing, read the help, and then start over. Having an external help utility or help file can be great, especially if you have two monitors: you can position PowerShell on one monitor and the help on the second. I mentioned a few external help utilities and files, all of which are free. Download one (or all) of them and set it up on your computer to use for the remainder of this book.

# The pipeline: connecting commands

In chapter 2, you saw that running commands in PowerShell is basically the same as running commands in any other shell: you type a command name, give it some parameters, and hit Return. What makes PowerShell so special isn't the way it runs commands, but rather the way it allows multiple commands to be connected to each other in powerful, one-line sequences.

## 4.1 Connect one command to another: less work for you!

PowerShell connects commands to each other in something called a *pipeline*. The pipeline is simply a way for one command to pass, or pipe, its output to another command, so that the second command has something to work with.

You've already seen this in action when you run something like `Dir | More`. You're piping the output of the `Dir` command into the `More` command; the `More` command takes that directory listing and displays it one page at a time. PowerShell takes that same piping concept and extends it to much greater effect. In fact, PowerShell's use of a pipeline may seem similar, at first, to how Unix and Linux shells work. Don't be fooled, though. As you'll come to realize over the next few chapters, PowerShell's pipeline implementation is much richer and more modern.

## 4.2 Exporting to a CSV or XML file

Run a simple command. Here are a few suggestions:

- `Get-Process` (or `Ps`)
- `Get-Service` (or `Gsv`)
- `Get-EventLog Security -newest 100`

I chose these because they're easy, straightforward commands; in parentheses, I've given you aliases for `Get-Process` and `Get-Service`. For `Get-EventLog`, I also specified its mandatory parameter as well as the `-newest` parameter (so the command wouldn't take too long to execute).

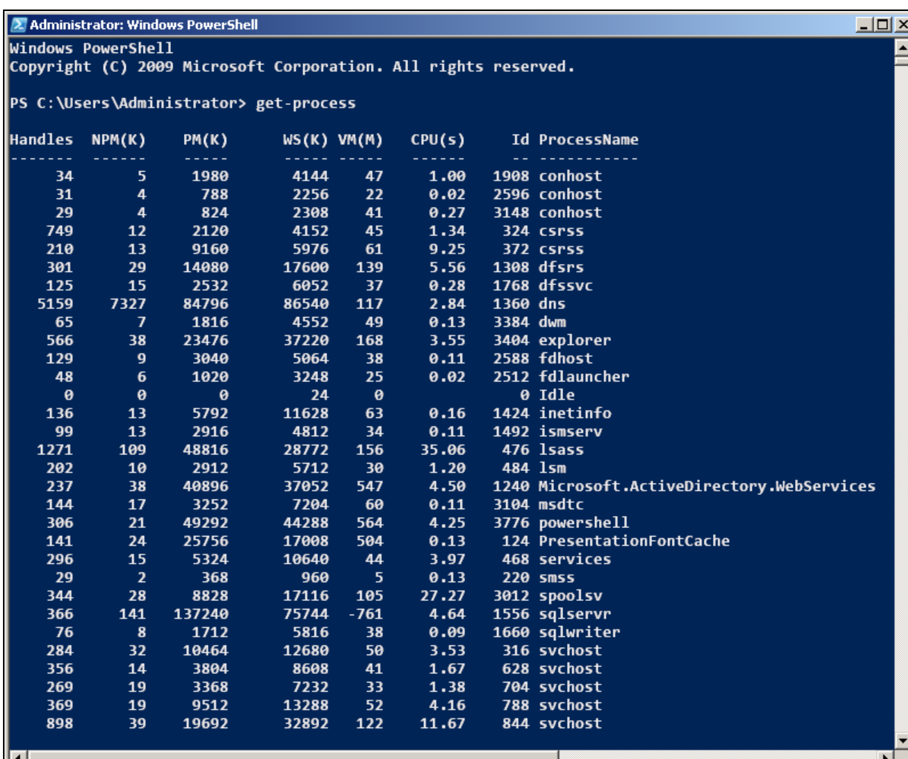
> **TRY IT NOW** Go ahead and choose one of these commands to work with. I'll use `Get-Process` for the following examples; you can stick with one of these, or switch between them to see the differences in the results.

What do you see? When I run `Get-Process`, a table (shown in figure 4.1) with several columns of information appears on the screen.

It's great to have that information on the screen, but that isn't all I might want to do with the information. For example, if I wanted to make some charts and graphs of memory and CPU utilization, I might want to export the information into a CSV (comma-separated values) file that could be read into an application like Microsoft Excel.

That's where the pipeline, and a second command, come in handy:

```
Get-Process | Export-CSV procs.csv
```

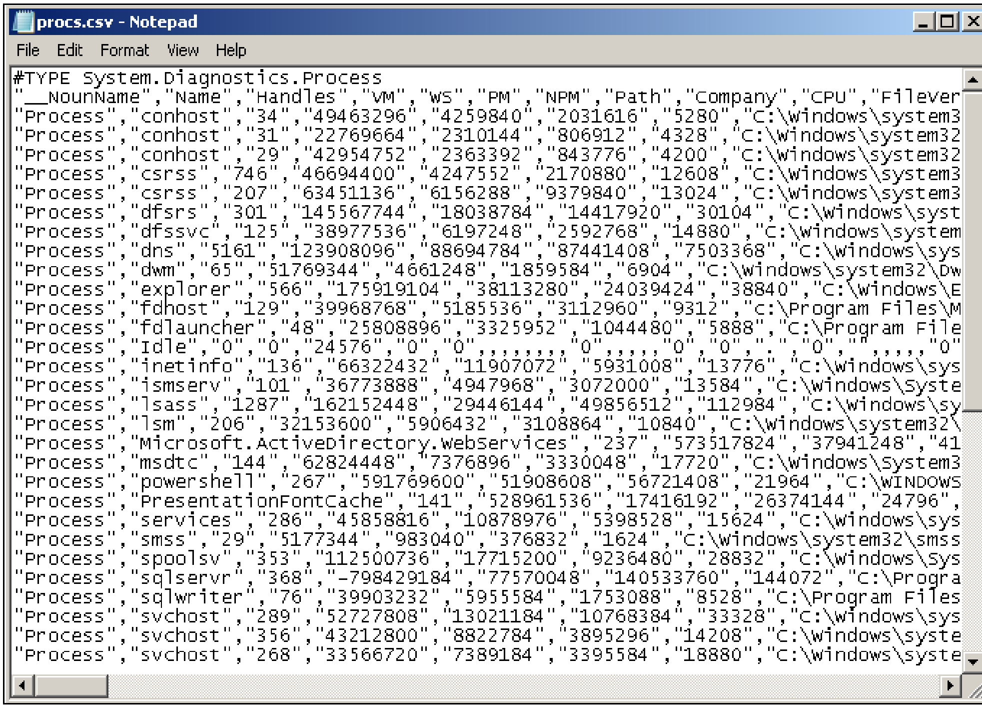


```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\Administrator> get-process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     34       5     1980       4144    47     1.00   1908 conhost
     31       4      788       2256    22     0.02   2596 conhost
     29       4      824       2308    41     0.27   3148 conhost
    749      12     2120       4152    45     1.34    324 csrss
    210      13     9160       5976    61     9.25    372 csrss
    301      29    14080      17600   139     5.56   1308 dfsrs
    125      15     2532       6052    37     0.28   1768 dfssvc
   5159    7327    84796      86540   117     2.84   1360 dns
     65       7     1816       4552    49     0.13   3384 dwm
    566      38    23476      37220   168     3.55   3404 explorer
    129       9     3040       5064    38     0.11   2588 fdhost
     48       6     1020       3248    25     0.02   2512 fdlauncher
      0       0        0         24     0              0 Idle
    136      13     5792      11628    63     0.16   1424 inetinfo
     99      13     2916       4812    34     0.11   1492 ismserv
   1271     109    48816      28772   156    35.06    476 lsass
    202      10     2912       5712    30     1.20    484 lsm
    237      38    40896      37052   547     4.50   1240 Microsoft.ActiveDirectory.WebServices
    144      17     3252       7204    60     0.11   3104 msdtc
    306      21    49292      44288   564     4.25   3776 powershell
    141      24    25756      17008   504     0.13    124 PresentationFontCache
    296      15     5324      10640    44     3.97    468 services
     29       2      368        960     5     0.13    220 smss
    344      28     8828      17116   105    27.27   3012 spoolsv
    366     141   137240      75744  -761     4.64   1556 sqlservr
     76       8     1712       5816    38     0.09   1660 sqlwriter
    284      32    10464      12680    50     3.53    316 svchost
    356      14     3804       8608    41     1.67    628 svchost
    269      19     3368       7232    33     1.38    704 svchost
    369      19     9512      13288    52     4.16    788 svchost
    898      39    19692      32892   122    11.67    844 svchost
```

**Figure 4.1 The output of `Get-Process` is a table with several columns of information.**



procs.csv - Notepad
File Edit Format View Help
```
#TYPE System.Diagnostics.Process
"__NounName","Name","Handles","VM","WS","PM","NPM","Path","Company","CPU","FileVer
"Process","conhost","34","49463296","4259840","2031616","5280","C:\Windows\system3
"Process","conhost","31","22769664","2310144","806912","4328","C:\Windows\system32
"Process","conhost","29","42954752","2363392","843776","4200","C:\Windows\system32
"Process","csrss","746","46694400","4247552","2170880","12608","C:\Windows\system3
"Process","csrss","207","63451136","6156288","9379840","13024","C:\Windows\system3
"Process","dfsrs","301","145567744","18038784","14417920","30104","C:\Windows\syst
"Process","dfssvc","125","38977536","6197248","2592768","14880","C:\Windows\system
"Process","dns","5161","123908096","88694784","87441408","7503368","C:\Windows\sys
"Process","dwm","65","51769344","4661248","1859584","6904","C:\Windows\system32\Dw
"Process","explorer","566","175919104","38113280","24039424","38840","C:\Windows\E
"Process","fdhost","129","39968768","5185536","3112960","9312","c:\Program Files\M
"Process","fdlauncher","48","25808896","3325952","1044480","5888","C:\Program File
"Process","Idle","0","0","24576","0","0",,,,,,,,"0",,,,,"0","0",".","0","",,,,,"0"
"Process","inetinfo","136","66322432","11907072","5931008","13776","C:\Windows\sys
"Process","ismserv","101","36773888","4947968","3072000","13584","C:\Windows\Syste
"Process","lsass","1287","162152448","29446144","49856512","112984","C:\Windows\sy
"Process","lsm","206","32153600","5906432","3108864","10840","C:\Windows\system32\
"Process","Microsoft.ActiveDirectory.WebServices","237","573517824","37941248","41
"Process","msdtc","144","62824448","7376896","3330048","17720","C:\Windows\System3
"Process","powershell","267","591769600","51908608","56721408","21964","C:\WINDOWS
"Process","PresentationFontCache","141","528961536","17416192","26374144","24796",
"Process","services","286","45858816","10878976","5398528","15624","C:\Windows\sys
"Process","smss","29","5177344","983040","376832","1624","C:\Windows\system32\smss
"Process","spoolsv","353","112500736","17715200","9236480","28832","C:\Windows\Sys
"Process","sqlservr","368","-798429184","77570048","140533760","144072","C:\Progra
"Process","sqlwriter","76","39903232","5955584","1753088","8528","C:\Program Files
"Process","svchost","289","52727808","13021184","10768384","33328","C:\Windows\sys
"Process","svchost","356","43212800","8822784","3895296","14208","C:\Windows\syste
"Process","svchost","268","33566720","7389184","3395584","18880","C:\Windows\syste
```

**Figure 4.2 Viewing the exported CSV file in Windows Notepad**

Just like piping `Dir` to `More`, I've piped my processes to `Export-CSV`. That second cmdlet has a mandatory positional parameter that I've used to specify the output filename. Because `Export-CSV` is a native PowerShell cmdlet, it knows how to translate the table normally generated by `Get-Process` into a normal CSV file.

Go ahead and open the file in Windows Notepad to see the results, as shown in figure 4.2:

```
Notepad procs.csv
```

The first line of the file will be a comment, preceded by a # sign, and it identifies the kind of information that's included in the file. In my example, it's `System.Diagnostics.Process`, which is the under-the-hood name that Windows uses to identify the information related to a running process. The second line will be column headings, and the subsequent lines will list the information for the various processes running on the computer.

You can pipe the output of almost any `Get-` cmdlet to `Export-CSV` and get excellent results. You may also notice that the CSV file contains a great deal more information than what is normally shown on the screen. That's deliberate. The shell knows it couldn't possibly fit all of that information on the screen, so it uses a configuration file, supplied by Microsoft, to select the most important information for on-screen display. In later chapters, I'll show you how to override that configuration to display whatever you want.

Once the information is saved into a CSV file, you could easily email it to a colleague and ask them to view it from within PowerShell. They'd simply import the file:

```
Import-CSV procs.csv
```

The shell would read in the CSV file and display the process information. It wouldn't be based on live information, of course, but it would be a snapshot from the exact point in time when you created the CSV file.

What if CSV files aren't what you need? PowerShell also has an `Export-CliXML` cmdlet, which creates a generic command-line interface (CLI) Extensible Markup Language (XML) file. CliXML is unique to PowerShell, but it can be read by any program capable of understanding XML. There's also a matching `Import-CliXML` cmdlet. Both cmdlets, like `Import-CSV` and `Export-CSV`, expect a filename as a mandatory parameter.

> **TRY IT NOW** Try exporting something, such as services, processes, or event log entries, to a CliXML file. Make sure you can re-import the file, and try opening the resulting file in Notepad and Internet Explorer to see how each of those applications displays the information.

Does PowerShell include any other import or export commands? You could find out by using the `Get-Command` cmdlet and specifying a `-verb` parameter with either `Import` or `Export`.

> **TRY IT NOW** See if PowerShell comes with any other import or export cmdlets. You may want to repeat this check after you load new commands into the shell, which is something you'll do in the next chapter.

Both CSV and CliXML files can be useful for persisting snapshots of information, sharing those snapshots with others, and reviewing those snapshots at a later time. In fact, let's look at one more cmdlet that has a great way of using those snapshots: `Compare-Object`. It has an alias, `Diff`, which I'll use.

First, run `help diff` and read the help for this cmdlet. There are three parameters in particular that I want you to pay attention to: `-ReferenceObject`, `-DifferenceObject`, and `-Property`.

`Diff` is designed to take two sets of information and compare them to each other. For example, imagine that you ran `Get-Process` on two different computers that were sitting side by side. The computer that's configured just the way you want is on the left and is the *reference computer.* The computer on the right might be exactly the same, or it might be somewhat different; it's the *difference computer.* After running the command on each, you'll be staring at two tables of information, and your job is to figure out if there are any differences between the two.

Because these are processes that you're looking at, you're always going to see differences in things like CPU and memory utilization numbers, so we'll ignore those columns. In fact, just focus on the Name column, because we really want to see if the difference computer contains any additional, or any fewer, processes than the reference computer. It might take you a while to compare all the process names from both tables, but you don't have to—that's exactly what `Diff` will do for you.

Let's say you sit down at the reference computer and run this:

```
Get-Process | Export-CliXML reference.xml
```

I prefer CliXML over CSV for comparisons like this, because CliXML can hold more information than a flat CSV file. You then transport that XML file over to the difference computer, and run this:

```
Diff -reference (Import-CliXML reference.xml)
➥ -difference (Get-Process) -property Name
```

This is a bit tricky, so I'll walk you through what's happening:

- Just like in math, parentheses in PowerShell control the order of execution. In this example, they force `Import-CliXML` and `Get-Process` to run before `Diff` runs. The output from `Import-CLI` is fed to the `-reference` parameter, and the output from `Get-Process` is fed to the `-difference` parameter.

  Actually, those parameter names are `-referenceObject` and `-differenceObject`; keep in mind that you can abbreviate parameter names by typing just enough of their names for the shell to be able to figure out which one you meant. In this case, `-reference` and `-difference` are more than enough to uniquely identify these parameters. I probably could have shortened them even further to something like `-ref` and `-diff`, and the command would still have worked.
- Rather than comparing the two complete tables, `Diff` focuses on the `Name`, because I gave it the `-property` parameter. If I hadn't, it would think that every process is different because the values of columns like VM, CPU, and PM are always going to be different.
- The result will be a table telling you what's different. Every process that's in the reference set, but not in the difference set, will have a `<=` indicator (indicating that the process is only present on the left side). If a process is on the difference computer but not the reference computer, it'll have a `=>` indicator instead. Processes that match across both sets won't be included in the `Diff` output.

**TRY IT NOW** Go ahead and try this. If you don't have two computers, start by exporting your current processes to a CliXML file as I've shown above. Then, start some additional processes like Notepad, Windows Paint, Solitaire, or whatever. Your computer will then be the difference computer (on the right), whereas the CliXML file will still be the reference set (on the left).

Here's the output from my test:

```
PS C:\> diff -reference (import-clixml reference.xml) -difference (get
-process) -property name

name                               SideIndicator
----                               -------------
calc                               =>
mspaint                            =>
notepad                            =>
conhost                            <=
powershell_ise                     <=
```

This is a really useful management trick. If you think of those reference CliXML files as configuration baselines, you can compare any current computer to that baseline and get a difference report. Throughout this book, you'll discover more cmdlets that can retrieve management information, all of which can be piped into a CliXML file to become a baseline. You can quickly build a collection of baseline files for services, processes, operating system configuration, users and groups, and much more, and then use those at any time to compare the current state of a system to its baseline.

> **TRY IT NOW** Just for fun, try running the `Diff` command again, but leave off the `-property` parameter entirely. See the results? Every single process is listed, because values like PM, VM, and so forth have all changed, even though they're the same processes. The output also isn't as useful, because it simply displays the process's type name and process name.

By the way, you should know that `Diff` generally doesn't do well at comparing text files. Although other operating systems and shells have a `Diff` command that's explicitly intended for comparing text files, PowerShell's `Diff` command works very differently. You'll see just how differently in this chapter's concluding lab.

> **NOTE** If it seems like you're using `Get-Process`, `Get-Service`, and `Get-EventLog` a lot, well, that's on purpose. I'm guaranteed that you have access to those cmdlets because they're native to PowerShell and don't require an add-in like Exchange or SharePoint. That said, the skills you're learning will apply to every cmdlet you ever need to run, including those that ship with Exchange, SharePoint, SQL Server, and other server products. Chapter 26 will go into that idea in more detail, but for now, focus on *how* to use these cmdlets rather than what the cmdlets are accomplishing. I'll work in some other representative cmdlets at the right time.

## 4.3 Piping to a file or printer

Whenever you have nicely formatted output—like the tables generated by `Get-Service` or `Get-Process`—you may want to preserve that in a file, or even on paper. Normally, cmdlet output is directed to the screen, which PowerShell refers to as the *Host*. You can change where that output goes. In fact, I've already showed you one way to do so:

```
Dir > DirectoryList.txt
```

That's a shortcut added to PowerShell to provide syntactic compatibility with the older Cmd.exe shell. In reality, when you run that command, here's what PowerShell does under the hood:

```
Dir | Out-File DirectoryList.txt
```

You can run that same command on your own, instead of using the `>` syntax. Why would you do so? `Out-File` also provides additional parameters that let you specify alternative character encodings (such as UTF8 or Unicode), append content to an existing file, and so forth. By default, the files created by `Out-File` are 80 columns wide, so sometimes PowerShell might alter command output to fit within 80 characters. That alteration might make the file's contents appear different than when you run the same command on the screen. Read its help file and see if you can spot a parameter of `Out-File` that would let you change the output file width to something other than 80 characters.

> **TRY IT NOW** Don't look here—open up that help file and see what you can find. I guarantee you'll spot the right parameter in a few moments.

PowerShell has a variety of `Out-` cmdlets. One is called `Out-Default`, and that's the one the shell uses when you don't specify a different `Out-` cmdlet. If you run this,

```
Dir
```

you're technically running this,

```
Dir | Out-Default
```

even if you don't realize it. `Out-Default` does nothing more than direct content to `Out-Host`, so you're really running this,

```
Dir | Out-Default | Out-Host
```

without realizing it. `Out-Host` is what handles getting information displayed on the screen. What other `Out-` cmdlets can you find?

> **TRY IT NOW** See what other `Out-` cmdlets you can discover. One way would be to use the `Help` command, using wildcards, such as `Help Out*`. Another would be to use `Get-Command` the same way, such as `Get-Command Out*`. Or, you could specify the `-verb` parameter: `Get-Command -verb Out`. What do you come up with?

`Out-Printer` is probably one of the most useful of the remaining `Out-` cmdlets. `Out-GridView` is also neat; it does require, however, that you have Microsoft .NET Framework v3.5 and the Windows PowerShell ISE installed, which isn't the case by default on server operating systems.

If you do have those installed, try running `Get-Service | Out-GridView` to see what happens. `Out-Null` and `Out-String` have specific uses that we won't get into right now, but you're welcome to read their help files and look at the examples included in those files.

## 4.4 Converting to HTML

Want to produce HTML reports? Easy: pipe your command to `ConvertTo-HTML`. This command produces well-formed, generic HTML that will display in any web browser. It's plain-looking, but you can reference a Cascading Style Sheet (CSS) to specify prettier formatting if desired. Notice that this doesn't require a filename:

```
Get-Service | ConvertTo-HTML
```

**TRY IT NOW** Make sure you run that command yourself—I want you to see what it does before you proceed.

In the PowerShell world, the verb `Export` implies that you're taking data, converting it to some other format, and saving that other format in some kind of storage, such as a file. The verb `ConvertTo` implies only a portion of that process: the conversion to a different format, but not saving it into a file. So when you ran the preceding command, you got a screen full of HTML, which probably isn't what you want. Stop for a second: can you think of how you'd get that HTML into a text file on disk?

**TRY IT NOW** If you can think of a way, go ahead and try it before you read on.

This command would do the trick:

```
Get-Service | ConvertTo-HTML | Out-File services.html
```

See how connecting more and more commands allows you to have increasingly powerful command lines? Each command handles a single step in the process, and the entire command line as a whole accomplishes a useful task.

PowerShell ships with other `ConvertTo-` cmdlets, including `ConvertTo-CSV` and `ConvertTo-XML`. As with `ConvertTo-HTML`, these don't create a file on disk; they translate command output into CSV or XML, respectively. You could pipe that converted output to `Out-File` to then save it to disk, although it would be shorter to use `Export-CSV` or `Export-CliXML`, because those do both the conversion and the saving.

### Above and beyond

Time for a bit more useless background information, although, in this case, it's the answer to a question that a lot of students often ask me: why would Microsoft provide both `Export-CSV` and `ConvertTo-CSV`, as well as two nearly identical cmdlets for XML? In certain advanced scenarios, you might not want to save the data to a file on disk. For example, you might want to convert data to XML and then transmit it to a web service, or some other destination. By having distinct `ConvertTo-` cmdlets that don't save to a file, you have the flexibility of doing whatever you want.

## 4.5 Using cmdlets to kill processes and stop services

Exporting and converting aren't the only reasons you might want to connect two commands together. For example, consider—but *please do not run*—this command:

```
Get-Process | Stop-Process
```

Can you imagine what that command would do? I'll tell you: crash your computer. It would retrieve every process and then start trying to end each one of them. It would get to a critical process, like the Local Security Authority, and your computer would probably crash with the famous Blue Screen of Death (BSOD). If you're running PowerShell inside of a virtual machine and want to have a little fun, go ahead and try running that command.

The point is that cmdlets with the same noun (in this case, `Process`) can often pass information between each other. Typically, you would specify the name of a specific process rather than trying to stop them all:

```
Get-Process -name Notepad | Stop-Process
```

Services offer something similar: the output from `Get-Service` can be piped to cmdlets like `Stop-Service`, `Start-Service`, `Set-Service`, and so forth. As you might expect, there are some specific rules about which commands can connect to each other. For example, if you look at a command sequence like `Get-ADUser | New-SQL-Database`, you would probably not expect it to do anything sensible (although it might well do something nonsensical). In chapter 7, we'll dive into the rules that govern how commands can connect to each other.

There is one more thing I'd like you to know about cmdlets like `Stop-Service` and `Stop-Process`. These cmdlets modify the system in some fashion, and all cmdlets that modify the system have an internally defined *impact level*. This impact level is set by the cmdlet's creator, and it can't be changed. The shell has a corresponding `$ConfirmPreference` setting, which is set to `High` by default. You can see your shell's setting by typing the setting name, like this:

```
PS C:\> $confirmpreference
High
```

Here's how it works: When a cmdlet's internal impact level is equal to or higher than the shell's `$ConfirmPreference` setting, the shell will automatically ask, "Are you sure?" when the cmdlet does whatever it's trying to do. In fact, if you tried the crash-your-computer command, earlier, you probably were asked, "Are you sure?" for each process. When a cmdlet's internal impact level is less than the shell's `$ConfirmPreference`, you don't automatically get the "Are you sure?" prompt.

You can, however, force the shell to ask you if you're sure:

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

Just add the `-confirm` parameter to the cmdlet. This should be supported by any cmdlet that makes some kind of change to the system, and it'll show up in the help file for the cmdlet if it's supported.

A similar parameter is `-whatif`. This is supported by any cmdlet that supports `-confirm`. The `-whatif` parameter isn't triggered by default, but you can specify it whenever you want to:

```
PS C:\> get-process | stop-process -whatif
What if: Performing operation "Stop-Process" on Target "conhost (1920)
".
What if: Performing operation "Stop-Process" on Target "conhost (1960)
".
What if: Performing operation "Stop-Process" on Target "conhost (2460)
".
What if: Performing operation "Stop-Process" on Target "csrss (316)".
```

It tells you what the cmdlet would have done, without actually letting the cmdlet do it. It's a useful way to preview what a potentially dangerous cmdlet would have done to your computer, to make certain that you want to do that.

## 4.6 Lab

I've kept this chapter's text a bit shorter because some of the examples I showed you probably took a bit longer to complete, and because I want you to spend a bit more time completing the following hands-on exercises. If you haven't already completed all of the "Try it now" tasks in the chapter, I strongly recommend that you do so before tackling these tasks:

1. Create a CliXML reference file for the services on your computer. Then, change the status of some non-essential service like BITS (stop it if it's already started; start it if it's stopped on your computer). Finally, use `Diff` to compare the reference CliXML file to the current state of your computer's services. You'll need to specify more than the `Name` property for the comparison—does the `-property` parameter of `Diff` accept multiple values? How would you specify those multiple values?
2. Create two similar, but different, text files. Try comparing them using `Diff`. To do so, run something like this: `Diff -reference (Get-Content File1.txt) -difference (Get-Content File2.txt)`. If the files have only one line of text that's different, the command should work. If you add a bunch of lines to one file, the command may stop working. Try experimenting with the `Diff` command's `-syncWindow` parameter to see if you can get the command working again.
3. What happens if you run `Get-Service | Export-CSV services.csv | Out-File` from the console? Why does that happen?
4. Apart from getting one or more services and piping them to `Stop-Service`, what other means does `Stop-Service` provide for you to specify the service or services you want to stop? Is it possible to stop a service without using `Get-Service` at all?

5 What if you wanted to create a pipe-delimited file instead of a comma-separated file? You would still use the `Export-CSV` command, but what parameters would you specify?
6 Is there a way to eliminate the `#` comment line from the top of an exported CSV file? That line normally contains type information, but what if you wanted to omit that from a particular file?
7 `Export-CliXML` and `Export-CSV` both modify the system, because they can create and overwrite files. What parameter would prevent them from overwriting an existing file? What parameter would ask you if you were sure before proceeding to write the output file?
8 Windows maintains several regional settings, which include a default list separator. On U.S. systems, that separator is a comma. How can you tell `Export-CSV` to use the system's default separator, rather than a comma?

# 5 Adding commands

One of the primary strengths of PowerShell is its extensibility. As Microsoft continues to invest in PowerShell, they develop more and more commands for products like Exchange Server, SharePoint Server, the System Center family, SQL Server, and so on. Typically, installing the management tools for these products gives you both a graphical management console of some kind and one or more extensions for Windows PowerShell.

## 5.1 How one shell can do everything

I know that you're probably familiar with the graphical Microsoft Management Console (MMC), so let's use that as an example of how PowerShell works. The two work similarly when it comes to extensibility, in part because both the MMC and PowerShell are developed by the same Management Frameworks team within Microsoft.

When you open a new, blank MMC console, it's pretty useless. It can't really do anything, because the MMC has very little built-in functionality. To make it useful, you go to its File menu and select Add/Remove Snapins. In the MMC world, a *snap-in* is some tool like Active Directory Users and Computers, or DNS Management, or DHCP Administration, or something like that. You can choose to add as many snap-ins to your MMC as you like, and you can save the resulting console so that it's easier to re-open that same set of snap-ins in the future.

Where do snap-ins come from? Typically, you install the management tools associated with a product like Exchange Server or Forefront or System Center. Once you've done so, those products' snap-ins are listed on the Add/Remove Snapins dialog box within the MMC. Most products also install their own preconfigured

MMC console files, which do nothing but load up the basic MMC and preload a snap-in or two. You don't have to use those preconfigured consoles if you don't want to, because you can always open a blank MMC and load the exact snap-ins you want. For example, the preconfigured Exchange Server MMC console doesn't include the Active Directory Sites and Services snap-in, but you could easily create an MMC console that includes both Exchange and Sites and Services.

PowerShell works in almost exactly the same way. Install the management tools for a given product (the option to install management tools is usually included in a product's Setup menu—just try to install a product like Exchange Server on Windows 7, and the management tools will often be the only thing Setup offers). Doing so will give you any related PowerShell extensions, and it may even create a product-specific management shell.

## 5.2 About product-specific management shells

Those product-specific management shells have been a huge source of confusion. Let me clearly state that there is only one Windows PowerShell. There isn't a separate PowerShell for Exchange and Active Directory; it's all a single shell.

Let's take Active Directory as an example, because I'm hoping that you have access to a Windows Server 2008 R2 domain controller (even if it's running in a virtual machine as a standalone domain). Open the Start menu, go to Administrative Tools, and locate the Active Directory Module for Windows PowerShell. Right-click that item, and select Properties from the context menu. The first thing you should see is the Target, which should be this:

```
%windir%\system32\WindowsPowerShell\v1.0\powershell.exe
➥ -noexit -command import-module ActiveDirectory
```

See? This is running the standard PowerShell.exe application and giving it a command-line parameter to run a specific command: `Import-Module ActiveDirectory`. The result is a copy of the shell that has the ActiveDirectory module preloaded, but there's no reason in the world why you couldn't open the "normal" PowerShell and run that same command yourself to get the same functionality.

The same thing holds true for almost every product-specific "management shell" that you'll find: Exchange, SharePoint, you name it. Examine the properties of those Start menu shortcuts, and you'll find that they open the normal PowerShell.exe, and pass a command-line parameter to either import a module, add a snap-in, or load a preconfigured console file (and the console file is simply a list of snap-ins to load automatically).

SQL Server 2008 and SQL Server 2008 R2 are exceptions. Their "product-specific" shell, Sqlps, is a specially compiled version of PowerShell that will only run the SQL Server extensions. Properly called a *mini-shell*, this is an approach Microsoft tried for the first time in SQL Server. It has been unpopular, and the company won't be using that approach again.

You're not constrained to working with the prespecified extensions. Once you open the Exchange Management Shell, you could run `Import-Module ActiveDirectory`, and provided the ActiveDirectory module was present on your computer, you'd add the Active Directory functionality to that shell. You could also open a normal PowerShell console and manually add whatever extensions you like.

As I said, this has been a huge point of confusion for folks, some of whom believed there were multiple versions of PowerShell that could not cross-utilize each others' functionality. I even got into an argument in my blog (http://windowsitpro.com/go/DonJonesPowerShell) over it at one point and had to ask half the PowerShell team to step in and back me up! So trust me: you can have all the functionality you want inside a single shell, and the product-specific shell shortcuts in the Start menu don't in any way limit you or imply that there are special versions of PowerShell for those products.

## 5.3 Extensions: finding and adding snap-ins

There are two kinds of extensions for PowerShell v2: modules and snap-ins. We'll look at snap-ins first.

The proper name for a snap-in is *PSSnapin*, which distinguishes these from snap-ins for the graphical MMS. PSSnapins were first created for PowerShell v1. A PSSnapin generally consists of one or more DLL files, accompanied by additional XML files that contain configuration settings and help text. PSSnapins have to be installed and registered in order for PowerShell to know they exist.

You can find a list of available snap-ins by running `Get-PSSnapin -registered` from within PowerShell. On my computer, which is a domain controller that happens to have SQL Server 2008 installed, I see this:

```
PS C:\> get-pssnapin -registered

Name        : SqlServerCmdletSnapin100
PSVersion   : 2.0
Description : This is a PowerShell snap-in that includes various SQL
              Server cmdlets.

Name        : SqlServerProviderSnapin100
PSVersion   : 2.0
Description : SQL Server Provider
```

> **TRY IT NOW** You should follow along with everything in this chapter, running the same commands in your own copy of PowerShell. I won't add a "Try it now" reminder for each command that I run, but I'll be expecting you to follow along.

That tells me that I have two snap-ins installed and available, but not loaded. You can see a list of loaded snap-ins by just running `Get-PSSnapin`. That list will include all of the core, automatically loaded snap-ins that contain PowerShell's native functionality.

To load a snap-in, run `Add-PSSnapin` and specify the name of the snap-in:

```
PS C:\> add-pssnapin sqlservercmdletsnapin100
```

As is often the case in PowerShell, you don't need to worry about getting upper- and lowercase letters correct. The shell won't care.

Once a snap-in is loaded, you'll want to figure out what it added to the shell. A PS-Snapin can add cmdlets, PSDrive providers, or both to the shell. To find out what cmdlets were added, use `Get-Command` (or its alias, `Gcm`):

```
PS C:\> gcm -pssnapin sqlservercmdletsnapin100

CommandType     Name                        Definition
-----------     ----                        ----------
Cmdlet          Invoke-PolicyEvaluation     Invoke-PolicyEvaluation...
Cmdlet          Invoke-Sqlcmd               Invoke-Sqlcmd [[-Query]...
```

Here, I've specified that only the commands from the SqlServerCmdletSnapin100 be included in the output, and only two were listed. Yes, that's all SQL Server adds in that snap-in, but one of those is capable of executing Transact-SQL (T-SQL) commands! Because you can accomplish almost anything in SQL Server by executing a T-SQL command, the `Invoke-Sqlcmd` cmdlet makes it possible to do almost anything you might need to do in SQL Server.

To see if the snap-in added any new PSDrive providers, run `Get-PSProvider`. You can't specify a snap-in with this cmdlet, so you'll have to be familiar with the providers that were already there, and scan through the list manually to spot anything new. Here are my results:

```
PS C:\> get-psprovider

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

Doesn't look like anything new. I shouldn't be surprised, because the snap-in I loaded was named SqlServerCmdletSnapin100. If you recall, my list of available snap-ins also included SqlServerProviderSnapin100, suggesting that the SQL Server team, for some reason, packaged their cmdlets and their PSDrive provider separately. Let's try adding the second one:

```
PS C:\> add-pssnapin sqlserverprovidersnapin100
PS C:\> get-psprovider

Name                 Capabilities               Drives
----                 ------------               ------
WSMan                Credentials                {WSMan}
Alias                ShouldProcess              {Alias}
Environment          ShouldProcess              {Env}
```

```
Name                 Capabilities             Drives         (continued)
----                 ------------             ------
FileSystem           Filter, ShouldProcess    {C, A, D}
Function             ShouldProcess            {Function}
Registry             ShouldProcess, Transa... {HKLM, HKCU}
Variable             ShouldProcess            {Variable}
Certificate          ShouldProcess            {cert}
SqlServer            Credentials              {SQLSERVER}
```

There we go! A SQLSERVER: drive has been added to my shell, powered by the SqlServer PSDrive provider. That means I could run `cd sqlserver:` to change to the SQL Server drive, and presumably start exploring databases and stuff.

## 5.4 Extensions: finding and adding modules

The second type of extension supported by PowerShell v2 (and not available in v1) is a *module*. Modules are designed to be a little more self-contained, and somewhat easier to distribute, but they work similarly to PSSnapins. You do need to know a bit more about them in order to find and use them.

Modules don't require advanced registration. Instead, PowerShell will automatically look in a certain set of paths to find modules. The `PSModulePath` environment variable defines the paths where modules are expected to live:

```
PS C:\> get-content env:psmodulepath
C:\Users\Administrator\Documents\WindowsPowerShell\Modules;C:\Windows
\system32\WindowsPowerShell\v1.0\Modules\
```

As you can see, there are two default locations: one in the operating system folder, where system modules live, and one in the Documents folder, where any personal modules can be added. You can also add a module from any other location, provided you know its full path.

There are a couple of ways to see what modules are available. One is to get a directory listing of those two paths. I'll just do the system path:

```
PS C:\> dir C:\windows\System32\WindowsPowerShell\v1.0\Modules

    Directory: C:\windows\System32\WindowsPowerShell\v1.0\Modules

Mode                LastWriteTime            Length Name
----                -------------            -----------
d---s        11/21/2009   9:58 AM            ActiveDirectory
d----         7/13/2009  10:41 PM            ADRMS
d---s         7/13/2009  10:41 PM            AppLocker
d----         7/13/2009  10:41 PM            BestPractices
d---s         7/13/2009  10:41 PM            BitsTransfer
d----        11/21/2009  10:08 AM            GroupPolicy
d----         7/13/2009  10:37 PM            PSDiagnostics
d----         7/13/2009  10:41 PM            ServerManager
d----         7/13/2009  10:41 PM            TroubleshootingPack
d----        11/21/2009  10:02 AM            WebAdministration
```

This doesn't help you locate any modules that might be installed in other locations, but hopefully if you install such a module, its documentation will help you figure out where it is. The preceding list shows the modules that come with Windows Server 2008 R2—or, at least, the modules installed on my server. Adding additional server roles or features may also add modules to support those roles and features, so it's worth checking this location any time you've installed something new.

Another way to get a list of available modules is to use `Get-Module`:

```
PS C:\> get-module -listavailable

ModuleType Name                          ExportedCommands
---------- ----                          ----------------
Manifest   ActiveDirectory               {}
Manifest   ADRMS                         {}
Manifest   AppLocker                     {}
Manifest   BestPractices                 {}
Manifest   BitsTransfer                  {}
Manifest   GroupPolicy                   {}
Manifest   PSDiagnostics                 {}
Manifest   ServerManager                 {}
Manifest   TroubleshootingPack           {}
Manifest   WebAdministration             {}
```

This list includes all modules installed in any path listed in the `PSModulePath` environment variable. These are the modules that the shell knows how to find. Any modules installed elsewhere won't be included in this list.

There are two ways to add a module, depending on whether or not the module is installed in one of the predefined paths. If the module is installed in one of those predefined paths, you use `Import-Module` and the module's name. You can then run `Get-Module`, with no parameters, to verify that the module loaded:

```
PS C:\> import-module activedirectory
PS C:\> get-module

ModuleType Name                          ExportedCommands
---------- ----                          ----------------
Manifest   activedirectory               {Set-ADOrganizationalUnit, Ge...
```

If the module is located elsewhere, you would need to specify the complete path to the module, such as C:\MyPrograms\Something\MyModule, rather than just the module name.

If you have a Start menu shortcut for a product-specific shell—say, SharePoint Server—and you don't know where that product installed its PowerShell module, open the properties for the Start menu shortcut. As I showed you earlier in this chapter, the `Target` property of the shortcut will contain the `Import-Module` command used to load the module, and that will show you the module name and path.

Once a module is loaded, you can find out what commands it added by using `Get-Command` again:

```
PS C:\> gcm -module activedirectory
```

```
CommandType     Name                          Definition
-----------     ----                          ----------
Cmdlet          Add-ADComputerServiceAc...    Add-ADComputerServiceAc...
Cmdlet          Add-ADDomainControllerP...    Add-ADDomainControllerP...
Cmdlet          Add-ADFineGrainedPasswo...    Add-ADFineGrainedPasswo...
Cmdlet          Add-ADGroupMember             Add-ADGroupMember [-Ide...
Cmdlet          Add-ADPrincipalGroupMem...    Add-ADPrincipalGroupMem...
Cmdlet          Clear-ADAccountExpiration     Clear-ADAccountExpirati...
```

This time, I used the `-module` parameter to specify the module name, limiting the list of commands to those that are included with the specified module.

Modules can also add PSDrive providers, and you would use the same technique as you did for PSSnapins to identify any new providers: run `Get-PSProvider`.

## 5.5 Command conflict and removing extensions

Take a close look at the commands I added for both SQL Server and Active Directory. Notice anything special about the commands' names?

Most PowerShell extensions—Exchange Server being a notable exception, because it was the first product to include a PowerShell extension, and they hadn't thought everything through at that point—add a short prefix to the noun portion of their command names. `Get-`*`AD`*`User`, for example, or `Invoke-`*`Sql`*`Cmd`. These prefixes may seem awkward, but they're designed to prevent command conflicts.

For example, suppose you loaded two modules that each contained a `Get-User` cmdlet. With two commands having the same name and being loaded at the same time, which one would PowerShell execute when you run `Get-User`? The last one loaded. But the other commands having the same name are not inaccessible. To specifically run either command, you would have to use a somewhat awkward naming convention that requires both the snap-in name and the command name. So if one `Get-User` came from a snap-in called MyCoolPowerShellSnapin, you'd have to run this:

```
MyCoolPowerShellSnapin\Get-User
```

That's an awful lot of typing, and it's why Microsoft suggests adding a product-specific prefix, like *AD* or *SQL*, to the noun of each command. Doing so helps prevent a conflict and helps make commands easier to identify and use.

If you do wind up with a conflict, you can always choose to remove one of the conflicting extensions. Simply run `Remove-PSSnapin` or `Remove-Module`, along with the snap-in or module name, to unload an extension.

## 5.6 Finding help on newly added commands

Once you have a list of newly added commands, you can start reading through their help. Microsoft-supplied snap-ins and modules usually come with help files, but third-party snap-ins or modules may not. (If they don't, you should definitely complain to the vendor—there's no reason not to provided integrated help.) Given that you already know how to get a list of command names from a snap-in or module, finding the help should be easy. For example, `Help Get-ADUser` will retrieve the help for that command.

Help information is stored as a sort of database, not as formatted text files. When you ask for help, PowerShell reads the help database for that command and dynamically constructs the help display that you see on the screen. That means all help files will have the same format, the same layout, and the same typographical conventions, keeping everything consistent. Everything that you learned about help files in chapter 3 will apply to the help for snap-ins and modules, including the ability to use the `-example` parameter to see examples of how to use the newly added commands.

## 5.7 Playing with Server Manager via command line!

Let's put your newfound knowledge to use. I'm going to assume that you're using a Windows Server 2008 R2 domain controller, and I'd like you to follow along with the commands I present in this section. More importantly, I want you to follow the process and the thinking that I'll explain, because this is exactly how I teach myself to use new commands without rushing out and buying a new book for every single product and feature that I run across. In the concluding lab for this chapter, I'll have you repeat this same process on your own, to learn about an entirely new set of commands.

My goal is to get an inventory of installed Windows roles and features, and to add a new role or feature. To begin with, I know that I'd normally use the Server Manager GUI. I'll start by looking for a snap-in or module that seems related to Server Manager. That requires me to run both `Get-PSSnapin -registered` and `Get-Module -listavailable`:

```
PS C:\> get-pssnapin -registered

Name        : SqlServerCmdletSnapin100
PSVersion   : 2.0
Description : This is a PowerShell snap-in that includes various SQL
              Server cmdlets.

Name        : SqlServerProviderSnapin100
PSVersion   : 2.0
Description : SQL Server Provider


PS C:\> get-module -list

ModuleType    Name                      ExportedCommands
----------    ----                      ----------------
Manifest      ActiveDirectory           {}
Manifest      ADRMS                     {}
Manifest      AppLocker                 {}
Manifest      BestPractices             {}
Manifest      BitsTransfer              {}
Manifest      GroupPolicy               {}
Manifest      PSDiagnostics             {}
Manifest      ServerManager             {}
Manifest      TroubleshootingPack       {}
Manifest      WebAdministration         {}

PS C:\>
```

I do see a ServerManager module in that list, so I'll start there. The next step is to get that module loaded into the shell. If it were a snap-in, I'd use `Add-PSSnapin`, but because it's a module, I'll use `Import-Module`:

```
PS C:\> import-module servermanager
```

Now I need to see what commands were added by that module. I'll use `Get-Command` to do so. If this had been a snap-in, I'd specify the `-pssnapin` parameter, but because it's a module, I'll use the `-module` parameter:

```
PS C:\> get-command -module servermanager

CommandType     Name                        Definition
-----------     ----                        ----------
Cmdlet          Add-WindowsFeature          Add-WindowsFeature [-Na...
Cmdlet          Get-WindowsFeature          Get-WindowsFeature [[-N...
Cmdlet          Remove-WindowsFeature       Remove-WindowsFeature [...
```

Well that's a short list! But it seems to have the functionality I'm after: commands to add, get, and remove Windows features. Hopefully that will include roles, and not just features. I always like to start with `Get-` commands, because they're non-destructive. I'll read the help first, just to be sure:

```
PS C:\>help get-windowsfeature
```

The help is pretty short for this one. It has an optional parameter, `-Name`, and a second optional parameter, `-logPath`. It also supports the common parameters, like all cmdlets, but I don't need to worry about any of those right now.

Because I don't see any mandatory parameters, I'll run the command without any parameters at all. If I missed a mandatory parameter, the shell will prompt me anyway. The output of the command is quite long, so I'll just include a portion of it here:

```
PS C:\> get-windowsfeature

Display Name                                            Name
------------                                            ----
[ ] Active Directory Certificate Services               AD-Certifi...
    [ ] Certification Authority                         ADCS-Cert-...
    [ ] Certification Authority Web Enrollment          ADCS-Web-E...
    [ ] Certificate Enrollment Web Service              ADCS-Enrol...
    [ ] Certificate Enrollment Policy Web Service       ADCS-Enrol...
[X] Active Directory Domain Services                    AD-Domain-...
    [X] Active Directory Domain Controller              ADDS-Domai...
    [ ] Identity Management for UNIX                    ADDS-Ident...
```

I can see which features and roles (it does include roles!) are installed, and which ones are available to install. The Name column appears to contain the official name of the role or feature. Unfortunately, I have my PowerShell window too narrow to display the full name (I did that so the output would be narrow enough to fit in this book), so I'll need to make the window a bit larger and run the command again.

I want to try to add a Windows feature. Scrolling down, I see that Telnet Client and Telnet Server are both available, with the official names `Telnet-Client` and `Telnet-Server`. Both of those features seem pretty harmless, so I'll experiment with them.

Reading the help for `Add-WindowsFeature`, I can see that it supports a mandatory `-Name` parameter that accepts more than one value. It also has an optional `-IncludeAllSubFeature` switch, and an optional `-logPath` parameter. Because this modifies the system, it supports `-confirm` and `-whatif`, which is nice. I also see a `-Restart` parameter, and I'm not sure what that does. I'll need more detail:

```
PS C:\>help add-windowsfeature -full
```

The detailed help for that parameter tells me that `-Restart` will restart the computer automatically if restarting is required. Well, that seems like a good idea, and I'm not testing on a production computer, so I'll go ahead and use that. I also see that the `-Concurrent` switch allows concurrent instances of the cmdlet to be running at the same time. I don't plan to do that, so I won't use that switch. The help also says that use of that switch "is not recommended," which is all the more reason for me to not use it!

Here we go:

```
PS C:\> add-windowsfeature -name telnet-client,telnet-server -restart
-whatif
What if: Checking if running in 'WhatIf' Mode.
What if: Performing operation "Add-WindowsFeature" on Target "[Telnet
Server] Telnet Server".
What if: Performing operation "Add-WindowsFeature" on Target "[Telnet
Client] Telnet Client".
What if: This server may need to be restarted after the installation c
ompletes.

Success  Restart Needed  Exit Code  Feature Result
-------  --------------  ---------  --------------
True     Maybe           Success    {}
```

You can see that I got nervous at the last second and added `-whatif`. The WhatIf output is telling me that it would have added Telnet Server and Telnet Client, and that I might need a restart. Well, okay, at least now I would know to warn anyone who might be using that server, or to schedule this for off-hours. Let's try again, this time without `-whatif`.

```
PS C:\>add-windowsfeature -name telnet-client,telnet-server -restart
```

A neat little progress bar pops up as the installation proceeds. I watch the progress bar inch its way up to 100 percent, and then the shell just sits there for a few minutes, thinking. Then it displays this:

```
Success  Restart Needed  Exit Code  Feature Result
-------  --------------  ---------  --------------
True     No              Success    {Telnet Server, Telnet Client}
```

So a restart wasn't necessary, and my server is still running. One last check:

```
PS C:\>get-windowsfeature
[ ] Active Directory Certificate Services               AD-Certifi...
    [ ] Certification Authority                         ADCS-Cert-...
    [ ] Certification Authority Web Enrollment          ADCS-Web-E...
    [ ] Certificate Enrollment Web Service              ADCS-Enrol...
    [ ] Certificate Enrollment Policy Web Service       ADCS-Enrol...
[X] Active Directory Domain Services                    AD-Domain-...
    [X] Active Directory Domain Controller              ADDS-Domai...
...
[ ] Subsystem for UNIX-based Applications               Subsystem-...
[X] Telnet Client                                       Telnet-Client
[X] Telnet Server                                       Telnet-Server
[ ] TFTP Client                                         TFTP-Client
```

I chopped out some of the output in the middle, so that you can see that the two Telnet features are now showing as installed. Perfect!

This is a great example of how to discover new functionality that matches what you already know how to do in the GUI. I found a module, loaded it, found the commands it includes, learned how to use those commands, and accomplished a task. This is PowerShell in action, not as a scripting language, but simply as a command-line shell. I could perform this same task even on Server Core, where PowerShell is available but the GUI Server Manager isn't.

## 5.8 Profile scripts: preloading extensions when the shell starts

Let's say you've opened PowerShell, and you've loaded several favorite snap-ins and modules. That requires you to run one command for each snap-in or module you want to load, which can take a few minutes of typing if there are several of them. Now you're done using the shell, so you close its window. The next time you open a shell window, all of your snap-ins and modules are gone, and you have to run all those commands again to load them back. Horrible! Surely there's a better way!

There are actually two better ways. The first way involves creating a *console file*, and this only works to memorize PSSnapins that are loaded—it won't work with any modules you may have loaded. Start by loading in all of the snap-ins you want, and then run this command:

```
Export-Console c:\myshell.psc
```

That creates a small XML file that lists the snap-ins you loaded into the shell.

Now, you'll want to create a new PowerShell shortcut somewhere. The target of that shortcut should be

```
%windir%\system32\WindowsPowerShell\v1.0\powershell.exe
➥ -noexit -psconsolefile c:\myshell.psc
```

When you use that shortcut to open a new PowerShell window, your console will load, and the shell will automatically add any snap-ins listed in that console file. Again, modules aren't included. So what do you do if you have a mix of snap-ins and modules, or if you have some modules that you always want loaded?

The answer is to use *profile scripts*. I've mentioned those before, and we're going to cover them in more detail in chapter 24, but for now here's how you can use them:

1. In your Documents folder, create a new folder called WindowsPowerShell (no spaces in that folder name).
2. In the newly created folder, use Notepad to create a file named profile.ps1. When you save the file in Notepad, be sure to enclose the filename in quotation marks: "profile.ps1". That will prevent Notepad from adding a .txt filename extension. If that .txt extension gets added, this trick won't work.
3. In that newly created text file, type your `Add-PSSnapin` and `Import-Module` commands, listing one command per line to load whatever snap-ins and modules you like.
4. Back in PowerShell, you'll need to enable script execution, which is disabled by default. There are some security consequences to this that we'll discuss in chapter 14, but for now I'm assuming that you're doing this in a standalone virtual machine, or on a standalone test computer, and that security is less of an issue. In the shell, run `Set-ExecutionPolicy RemoteSigned`. Note that the command will only work if you've run the shell as Administrator. It's also possible for a Group Policy object (GPO) to override this setting; you'll get a warning message if that's the case.
5. Assuming you haven't had any errors or warnings up to this point, close and reopen the shell. It will automatically load profile.ps1, execute your commands, and load your favorite snap-ins and modules for you.

**TRY IT NOW** Even if you don't have a favorite snap-in or module yet, creating this simple profile will be good practice. If nothing else, put the command `cd \` into the profile script, so that the shell always opens in the root of your system drive. But please don't do this on a computer that's part of your company's production network, because we haven't covered all of the security implications yet.

## 5.9 Common points of confusion

There's exactly one thing that I frequently see PowerShell newcomers do incorrectly when they start working with modules and snap-ins: they don't read the help. Specifically, they don't use the `-example` or `-full` switch when asking for help.

Frankly, looking at any built-in examples is the best way to learn how to use a command. Yes, it can be a bit daunting to scroll through a list of hundreds of commands (Exchange Server, for example, adds well over 400 new commands), but using `Help` and `Get-Command` with wildcards should make it easier to narrow down the list to whatever noun you think you're after. From there, *read the help*!

## 5.10 Lab

As always, I'm assuming that you have a Windows Server 2008 R2 computer or virtual machine to test with, and that it's configured as a domain controller. If you don't, revisit chapter 1, where I explain how you can download a mostly configured virtual machine from Microsoft. You'll just need to make that into a domain controller, and in chapter 1, I directed you to a tutorial on that.

For this lab, you only have one basic task: run a Best Practices Analyzer (BPA) report for Directory Services and DNS Server, the two models that should be present on your Windows Server 2008 R2 domain controller. BPA models can take a long time to run, so be patient while the shell thinks—don't get nervous and press Ctrl-C! Your final result should be an HTML file containing a table that lists the results of the BPA analysis.

That's all the help you get!

## 5.11 Ideas for on your own

Windows Server 2008 R2 contains numerous other modules that can help you automate administration. If you have some extra time, see if you can figure out how to read individual settings from a Group Policy object. There's a module that can do this, although using it can be a bit complicated. Remember that a GPO is the top-level item you'll want to work with, and then you'll want to dig into it to retrieve individual settings. Settings within a GPO come in two forms: registry values and registry preference values. The former are the GPO settings that a GPO can fully control.

To experiment, you may want to create a new GPO in your test domain, link it to an OU (you may even want to create an OU to link it to), and then change a few settings within the GPO. You can do all of that with the GUI Group Policy Management Console (GPMC), and then switch to PowerShell to try and query the individual settings. As always, rely on the examples in the help files to get you started.

This business of finding modules, locating their commands, reading the help, and experimenting with commands is the single most important thing you'll ever learn in PowerShell. In fact, if you want to stop reading and spend a few days experimenting with different commands, go right ahead. Teaching yourself new commands is absolutely the most valuable PowerShell skill you can acquire.

# Objects: just data by another name

We're going to do something a little different in this chapter. I find that PowerShell's use of objects can be one of its most confusing elements, but at the same time it's also one of the shell's most critical concepts, affecting everything you do in the shell. I've tried different explanations over the years, and I've settled on a couple that each work well for distinctly different audiences. So, if you have some programming experience and are comfortable with the concept of objects, I want you to skip to section 6.2. If you don't have a programming background, and haven't programmed or scripted with objects before, start with section 6.1 and read straight through.

## 6.1 What are objects?

Stop for a second and run `Get-Process` in PowerShell. You should see a table with several columns, but those columns barely scratch the surface of the wealth of information available about processes. Each process also has a machine name, a main window handle, a maximum working set size, an exit code and time, processor affinity information, and a great deal more. In fact, there are more than 60 pieces of information associated with a process. Why does PowerShell show so few of them?

The simple fact is that *most* of the things PowerShell can access offer more information than will comfortably fit on the screen. When you run any command, such as `Get-Process`, `Get-Service`, `Get-EventLog`, or anything, PowerShell constructs—entirely in memory—a table that contains all of the information about those items. In the case of `Get-Process`, that table consists of something like 67 columns, with one row for each process that's running on your computer. Each column contains a bit of information, such as virtual memory, CPU utilization, process name, process ID, and so on. Then, PowerShell looks to see if you have specified which of those columns you want to see. If you haven't (and I haven't shown you how, yet) then the shell looks up a configuration file provided by Microsoft and displays only those table columns that Microsoft thought you'd want to see.

One way to see all of the columns is to use `ConvertTo-HTML`:

```
Get-Process | ConvertTo-HTML | Out-File processes.html
```

That cmdlet doesn't bother filtering down the columns, so it produces an HTML file that contains all of them. That's one way to see the entire table.

In addition to all of those columns of information, each table row also has some actions associated with it. Those are things that the operating system can do to, or with, the process listed in that table row. For example, the operating system can close a process, kill it, refresh its information, or wait for the process to exit, among other things.

Any time you run a command that produces output, that output takes the form of a table in memory. When you pipe output from one command to another, like this,

```
Get-Process | ConvertTo-HTML
```

the entire table is passed through the pipeline. The table isn't filtered down to a smaller number of columns until every command has run.

Now for some terminology changes! PowerShell doesn't refer to this in-memory table as a "table." Instead, it uses these terms:

- *Object*—This is what I've been calling a "table row." It represents a single thing, like a single process or a single service.
- *Property*—This is what I called a "table column." It represents one piece of information about an object, like a process name, process ID, or service status.
- *Method*—This is what I called an "action." A method is related to a single object and makes that object do something, like killing a process or starting a service.
- *Collection*—This is the entire set of objects, or what I've been calling a "table."

If you ever find the following discussion on objects to be confusing, refer back to this four-point list. Always imagine a *collection* of objects as being a big in-memory table of information, with *properties* as the columns and individual *objects* as the rows.

## 6.2 Why PowerShell uses objects

One of the reasons that PowerShell uses objects to represent data is that, well, you have to represent data *somehow*, right? PowerShell could have chosen to store that data in a format like XML, or perhaps its creators could have chosen to use plain-text tables. There are some specific reasons why they didn't, however.

The first big reason is that Windows itself is an object-oriented operating system—or at least, most of the software that runs on Windows is object oriented. Choosing to structure data as a set of objects is easy, because most of the operating system lends itself to those structures.

Another reason to use objects is because they ultimately make things easier on you and give you more power and flexibility. For just a second, I want to pretend that PowerShell doesn't produce objects as the output of its commands. Instead, it produces simple text tables, which is what you probably thought it was doing in the first place. When you run a command like `Get-Process`, you're getting formatted text as the output:

```
PS C:\> get-process

Handles  NPM(K)    PM(K)      WS(K)   VM(M)   CPU(s)      Id   ProcessName
-------  ------    -----      -----   -----   ------      ----   -----------
     39       5     1876       4340      52    11.33    1920   conhost
     31       4      792       2260      22     0.00    2460   conhost
     29       4      828       2284      41     0.25    3192   conhost
    574      12     1864       3896      43     1.30     316   csrss
    181      13     5892       6348      59     9.14     356   csrss
    306      29    13936      18312     139     4.36    1300   dfsrs
    125      15     2528       6048      37     0.17    1756   dfssvc
   5159    7329    85052      86436     118     1.80    1356   dns
```

What if you wanted to do something else with this information? Perhaps you want to do something to all of the processes running Conhost. That means you're going to have to filter this list down a bit. In a Unix or Linux shell, you'd use a command like Grep, telling it, "Look at this text list for me. Keep only those rows where columns 58–64 contain the characters 'conhost.' Delete all of the other rows." The resulting list would contain just those processes you specified:

```
Handles  NPM(K)    PM(K)    WS(K)   VM(M)   CPU(s)     Id    ProcessName
-------  ------    -----    -----   -----   ------     ----    -----------
     39       5     1876     4340      52    11.33    1920    conhost
     31       4      792     2260      22     0.00    2460    conhost
     29       4      828     2284      41     0.25    3192    conhost
```

You'd then pipe that text to another command, perhaps telling it to extract the process ID from the list. "Go through this and get the characters from columns 52–56, but drop the first two rows." The result might be this:

```
1920
2460
3192
```

Finally, you'd pipe *that* text to yet *another* command, asking it to kill the processes (or whatever else you were trying to do) represented by those ID numbers.

This is, in fact, exactly how Unix and Linux administrators work. They spend a lot of time learning how to get very good at parsing text, using tools like Grep, Awk, and Sed, and becoming very proficient in the use of regular expressions, which make it easier for them to define text patterns that they want their computer to look for. Unix and Linux folks like programming languages like Perl because those languages contain rich text-parsing and text-manipulation functions.

There are, however, some problems with this text-based approach:

- You can spend more time messing around with text than doing your real job.

- If the output of a command changes—say, moving the ProcessName column to the start of the table—then you have to rewrite all of your commands, because they're all dependent on things like column positions.
- You have to become very proficient in languages and tools that parse text. Not because your job actually involves parsing text, but because parsing text is a means to an end.

PowerShell's use of objects helps to remove all of that text-manipulation overhead. Because objects work like a table in memory, you don't have to tell PowerShell which text column a piece of information is located at. Instead, you tell it the column name, and PowerShell knows exactly where to go to get that data. Regardless of how you arrange the final output on the screen or in a file, the in-memory table is always the same, so you never have to rewrite your commands because a column moved. You spend a lot less time on overhead tasks, and more time focusing on what it is you want to accomplish.

True, you do have to learn a few syntax elements that let you instruct PowerShell properly, but you'll have to learn a *lot* less than if you were working in a purely text-based shell.

## 6.3 Discovering objects: Get-Member

If objects are like a giant table in memory, and PowerShell only ever shows you a portion of that table on the screen, how can you see what else you have to work with? If you're thinking that you should use the `Help` command, then I'm glad, because I've certainly been pushing that down your throat in the previous few chapters! Unfortunately, you'd be wrong.

The help system only documents background concepts (in the form of the "about" help topics) and command syntax. To learn more about an object, you use a different command: `Get-Member`. You should become very comfortable using this command—so much so, in fact, that you start looking for a shorter way to type it. I'll give you that right now: the alias `Gm`.

You can use `Gm` after any cmdlet that normally produces some output. For example, you already know that running `Get-Process` produces some output on the screen. You can pipe it to `Gm`:

```
Get-Process | Gm
```

Whenever a cmdlet produces a collection of objects, as `Get-Process` does, the entire collection remains accessible until the end of the pipeline. It's not until every command has run that PowerShell filters down the columns of information that are to be displayed and creates the final text output that you see. Therefore, in the preceding example, `Gm` has complete access to all of the process objects' properties and methods, because they haven't been filtered down for display yet. `Gm` looks at each object and constructs a list of the objects' properties and methods. It looks a bit like this:

```
PS C:\> get-process | gm

   TypeName: System.Diagnostics.Process

Name                        MemberType         Definition
----                        ----------         ----------
Handles                     AliasProperty      Handles = Handlecount
Name                        AliasProperty      Name = ProcessName
NPM                         AliasProperty      NPM = NonpagedSystemMemo...
PM                          AliasProperty      PM = PagedMemorySize
VM                          AliasProperty      VM = VirtualMemorySize
WS                          AliasProperty      WS = WorkingSet
Disposed                    Event              System.EventHandler Disp...
ErrorDataReceived           Event              System.Diagnostics.DataR...
Exited                      Event              System.EventHandler Exit...
OutputDataReceived          Event              System.Diagnostics.DataR...
BeginErrorReadLine          Method             System.Void BeginErrorRe...
BeginOutputReadLine         Method             System.Void BeginOutputR...
CancelErrorRead             Method             System.Void CancelErrorR...
CancelOutputRead            Method             System.Void CancelOutput...
```

I've trimmed the list a bit because it's pretty long, but hopefully you get the idea.

> **TRY IT NOW** Don't take my word for it. This is the perfect time to start following along and running the same commands that I do, so you can see their full and complete output.

By the way, it may interest you to know that all of the properties, methods, and other things attached to an object are collectively called its *members*, as if the object itself were a country club and all of these properties and methods belonged to the club. That's where `Get-Member` takes its name from: it's getting a list of the objects' members. Of course, because the PowerShell convention is to use singular nouns, the cmdlet name is `Get-Member`, not "Get-Members."

## 6.4 Object attributes, or "properties"

When you examine the output of `Gm`, you'll notice several different kinds of properties:

- ScriptProperty
- Property
- NoteProperty
- AliasProperty

For your purposes, these are all the same. The only difference is how the values in those properties are obtained, but that's not something you need to worry about. To you, they're all "properties," and you'll use them the same way.

A property always contains a value. For example, the value of a process object's `ID` property might be `1234`, and the `Name` property of that object might have a value of `Notepad`. Properties describe something about the object: its status, its ID, its name, and so on. In PowerShell, properties are often read-only, meaning that you can't change the name of a service by assigning a new value to its `Name` property. You can, however, retrieve the name of a service by reading its `Name` property. Probably 90 percent of what you do in PowerShell will involve properties.

## 6.5 Object actions, or "methods"

Many objects support one or more methods, which, as I wrote earlier, are actions that you can direct the object to take. A process object has a `Kill` method, which terminates the process. Some methods require one or more input arguments that provide additional detail for that particular action, but this early in your PowerShell education you won't be running into any of those. In fact, you may spend months or even years working with PowerShell and never need to execute a single object method. That's because many of those actions are also provided by cmdlets.

For example, if I need to terminate a process, I have three ways that I could do so. One way would be to retrieve the object and then somehow execute its `Kill` method. Another way would be to use a couple of cmdlets:

```
Get-Process -Name Notepad | Stop-Process
```

I could also accomplish that by using a single cmdlet:

```
Stop-Process -name Notepad
```

My focus with this book is entirely on using PowerShell cmdlets to accomplish tasks. They provide the easiest, most administrator-centric, most task-focused way of accomplishing things. Using methods starts to edge into .NET Framework programming, which can be more complicated and can require a lot more background information. For that reason, you'll rarely—if ever—see me execute an object method in this book. In fact, my general philosophy at this point is, "If you can't do it with a cmdlet, then go back and use the GUI." You won't feel that way for your entire career, I promise, but for now it's a good way to stay focused on the "PowerShell way" of doing things.

### Above and beyond

You don't really need to know about them at this stage in your PowerShell education, but in addition to properties and methods, objects can also have *events*. An event is an object's way of notifying you that something happened to it. A process object, for example, can trigger its `Exited` event when the process ends. You can attach your own commands to those events, so that, for example, an email gets sent when a process exits. Working with events in this fashion is a pretty advanced topic, and it's beyond the scope of this book.

## 6.6 Sorting objects

Most PowerShell cmdlets produce objects in a deterministic fashion, which simply means that they tend to produce objects in the same order every time you run the command. Both services and processes, for example, are listed in alphabetical order by name. Event log entries tend to come out in chronological order. What if you want to change that?

For example, suppose I want to display a list of processes, with the biggest consumers of virtual memory (VM) at the top of the list, and the smallest consumers at the bottom. I would need to somehow re-order that list of objects based on the VM property. PowerShell provides a very simple cmdlet, `Sort-Object`, that does exactly that:

```
Get-Process | Sort-Object -property VM
```

> **TRY IT NOW** I'm hoping that you'll follow along and run the same commands that I am. I won't be pasting the output into the book because these tables are pretty long, but you'll get substantially the same thing on your screen if you're following along.

That isn't exactly what I wanted. It did sort on VM, but it did so in ascending order, with the largest values at the bottom of the list. Reading the help for `Sort-Object`, I see that it has a `-descending` parameter that should reverse the sort order. I also notice that the `-property` parameter is positional, so I don't need to type the parameter name. I'll also tell you that `Sort-Object` has an alias, `Sort`, so you can save yourself a bit of typing for the next try:

```
Get-Process | Sort VM -desc
```

I also abbreviated `-descending` to `-desc`, and the result is exactly what I was looking for. The `-property` parameter accepts multiple values (which I'm sure you saw in the help file, if you looked).

In the event that two processes are using the same amount of virtual memory, I'd like them sorted by process ID, and this will accomplish that:

```
Get-Process | Sort VM,ID -desc
```

As always, a comma-separated list is the way to pass multiple values to any parameter that supports them.

## 6.7 Selecting the properties you want

Another useful cmdlet is `Select-Object`. It accepts objects from the pipeline, and you can specify the properties that you would like displayed. This enables you to access properties that are normally filtered out by PowerShell's configuration rules, or to trim the list down to a few properties that interest you. This can be very useful when piping objects to `ConvertTo-HTML`, because that cmdlet usually builds a table containing every property. Compare the results of these two commands:

```
Get-Process | ConvertTo-HTML | Out-File test1.html

Get-Process | Select-Object -property Name,ID,VM,PM |
➥ Convert-ToHTML | Out-File test2.html
```

> **TRY IT NOW** Go ahead and run each of these commands separately, and then examine the resulting HTML files in Internet Explorer to see the differences.

Take a look at the help for `Select-Object` (or you can use its alias, `Select`). The `-property` parameter appears to be positional, which means I could shorten that last command to this:

```
Get-Process | Select Name,ID,VM,PM | ConvertTo-HTML | Out-File test3.html
```

Spend some time experimenting with `Select-Object`. In fact, try variations of this command, which allows the output to appear on the screen:

```
Get-Process | Select Name,ID,VM,PM
```

Try adding and removing different process object properties from that list and reviewing the results. How many properties can you specify and still get a table as the output? How many properties force PowerShell to format the output as a list rather than as a table?

## 6.8 Objects until the very end

The PowerShell pipeline always contains objects, right until the last command has been executed. At that time, PowerShell looks to see what objects are in the pipeline, and then looks at its various configuration files to see which properties will be used to construct the onscreen display. It also decides whether that display will be a table or a list, based on some internal rules and on its configuration files. (I'll explain more about those rules and configurations, and how you can modify them, in chapter 8.)

An important fact is that the pipeline can contain many different things over the course of a single command line. For the next few examples, I'm going to take a single command line and physically type it so that only one command appears on a single line of text. That'll make it a bit easier to explain what I'm talking about.

Here's the first one:

```
Get-Process |
Sort-Object VM -descending |
Out-File c:\procs.txt
```

In this example, I start by running `Get-Process`, which puts process objects into the pipeline. The next command is `Sort-Object`. That doesn't change what's in the pipeline; it just changes the order of the objects, so at the end of `Sort-Object`, the pipeline still contains processes. The last command is `Out-File`. Here, PowerShell has to produce output, so it takes whatever's in the pipeline—processes—and formats them according to its internal rule set. The results go into the specified file.

Next up is a more complicated example:

```
Get-Process |
Sort-Object VM -descending |
Select-Object Name,ID,VM
```

This starts off in the same way. `Get-Process` puts process objects into the pipeline. Those go to `Sort-Object`, which sorts them and puts the same process objects into the pipeline. `Select-Object` works a bit differently, though. You see, a process object always has the exact same members. In order to trim down the list of properties, `Select-Object` can't just remove the properties I don't want, because the result wouldn't be a process object anymore. Instead, `Select-Object` creates a new kind of custom object called a `PSObject`. It copies over the properties I do want from the process, resulting in a custom object being placed into the pipeline.

> **TRY IT NOW** Try running this three-cmdlet command line, keeping in mind that you should type the whole thing on a single line. Notice how the output is different from the normal output of `Get-Process`?

When PowerShell sees that it's reached the end of the command line, it has to decide how to lay out the text output. Because there are no longer any process objects in the pipeline, PowerShell won't use the default rules and configurations that apply to process objects. Instead, it looks for rules and configurations for a `PSObject`, which is what the pipeline now contains. Microsoft didn't provide any rules or configurations for `PSObject`s, because they're meant to be used for custom output. So, PowerShell takes its best guess and produces a table, on the theory that those three pieces of information will still probably fit in a table. The table isn't as nicely laid out as the normal output of `Get-Process`, though, because the shell lacks the additional configuration information needed to make a nicer-looking table.

You can use `Gm` to see the different objects that wind up in the pipeline. Remember, you can stick `Gm` in after any cmdlet that produces output:

```
Get-Process | Sort VM -descending | gm
Get-Process | Sort VM -descending | Select Name,ID,VM | gm
```

> **TRY IT NOW** Try running those two command lines separately, and notice the difference in the output.

Notice that, as part of the `Gm` output, it shows you the type name for the object it saw in the pipeline. In the first case, that was a `System.Diagnostics.Process` object, but in the second case the pipeline contains a different kind of object. Those new "selected" objects only contained the three properties specified—`Name`, `ID`, and `VM`—plus a couple of system-generated members.

Even `Gm` produces objects and places them into the pipeline! After running `Gm`, the pipeline no longer contained either process or the "selected" objects; it contained the type of object produced by `Gm`: `Microsoft.PowerShell.Commands.MemberDefinition`. You can prove that by piping the output of `Gm` to `Gm` itself:

```
Get-Process | Gm | Gm
```

> **TRY IT NOW** You'll definitely want to try this, and think hard about it to make sure it makes sense to you. You start with `Get-Process`, which puts process objects into the pipeline. Those go to `Gm`, which analyzes them and produces its own `MemberDefinition` objects. Those are then piped to `Gm`, which analyzes them and produces output that lists the members of a `MemberDefinition` object.

A real key in mastering PowerShell is learning to keep track of what kind of object is in the pipeline at any given point. `Gm` can help you do that, but sitting back and verbally walking yourself through the command line is also a good exercise that can help clear up confusion.

## 6.9 Common points of confusion

There are a few common mistakes that my classroom students tend to make as they get started with PowerShell. Most of these go away with a little bit of experience, but I'll direct your attention to them so that you can catch yourself if you start heading down the wrong path.

- Remember that the PowerShell help files don't contain information on objects' properties. You'll need to pipe the objects to `Gm` (`Get-Member`) to see a list of properties.
- Remember that you can add `Gm` to the end of any pipeline that normally produces results. A command line like `Get-Process -name Notepad | Stop-Process` doesn't normally produce results, so tacking `| Gm` onto the end won't produce anything either.
- Start paying attention to neat typing. Put a space on either side of every pipeline character, so that your command lines read like `Get-Process | Gm` and not `Get-Process|Gm`. That spacebar key is extra-large for a reason—use it!
- Always remember that the pipeline can contain different types of objects at each step. Think about what type of object is in the pipeline, and focus on what the next command will do to that *type* of object.

## 6.10 Lab

This chapter has probably covered more, and more difficult, new concepts than any chapter so far. Hopefully I was able to make it all make sense, but these exercises should help you cement everything. See if you can complete them all, and remember that there are companion videos and sample solutions at MoreLunches.com. Some of these tasks will draw on skills you learned in previous chapters, as a way of refreshing your memory and keeping you sharp.

1. Identify a cmdlet that will produce a random number.
2. Identify a cmdlet that will display the current date and time.
3. What type of object does the cmdlet from task #2 produce? (What is the *type name* of the object produced by the cmdlet?)
4. Using the cmdlet from task #2 and `Select-Object`, display only the current day of the week in a table like this:

```
DayOfWeek
---------
   Monday
```

5. Identify a cmdlet that will display information about installed hotfixes.

6 Using the cmdlet from task #5, display a list of installed hotfixes. Sort the list by the installation date, and display only the installation date, the user who installed the hotfix, and the hotfix ID.

7 Repeat task #6, but this time sort the results by the hotfix description, and include the description, the hotfix ID, and the installation date. Put the results into an HTML file.

8 Display a list of the 50 newest entries from the Security event log (you can use a different log, such as System or Application, if your Security log is empty). Sort the list so that the oldest entries appear first, and so that entries made at the same time are sorted by their index. Display the index, time, and source for each entry. Put this information into a text file (not an HTML file, just a plain text file).

# 7 The pipeline, deeper

So far, you've learned to be pretty effective with PowerShell's pipeline. Running commands like `Get-Process | Sort VM -desc | ConvertTo-HTML | Out-File procs.html` is pretty powerful, accomplishing in one line what used to take several lines of script. But you can do even better! In this chapter, we'll dig deeper into the pipeline and uncover some of its most powerful capabilities.

## 7.1 The pipeline: enabling power with less typing

One of the reasons I like PowerShell so much is that it enables me to be a more effective administrator without having to write complex scripts, like I used to have to do in VBScript. But the key to powerful one-line commands lies in the way the PowerShell pipeline works.

Let me be clear: you could skip this chapter and still be effective with PowerShell, but you would in most cases have to resort to VBScript-style scripts and programs. Although PowerShell's pipeline capabilities can be complicated, they're probably easier to learn than more-complicated programming skills, and by learning to really manipulate the pipeline, you can be much more effective without needing to write scripts.

The whole idea here is to get the shell to do more of your work for you, with as little typing as possible. I think you'll be surprised at how well the shell can do that!

## 7.2 Pipeline input ByValue, or why Stop-Service works

Let's start by looking at a command that you've seen earlier in this book:

```
Get-Service -Name Bits | Stop-Service
```

This command retrieves a single service, named *BITS* (it's the Background Intelligent Transfer Service, and I like to play with it in these examples because starting and stopping it won't wreck the operating system). It pipes that service to the `Stop-Service` cmdlet, which attempts to stop the service. Easy enough to understand, but why, exactly, does it work?

Let's start by carefully examining the output of `Get-Service`, by piping that output to `Get-Member` (or its alias, `Gm`):

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

   TypeName: System.ServiceProcess.ServiceController

Name                     MemberType       Definition
----                     ----------       ----------
Name                     AliasProperty    Name = ServiceName
RequiredServices         AliasProperty    RequiredServices = Service...
```

That output tells me that the `Get-Service` cmdlet is producing objects of the type `System.ServiceProcess.ServiceController`. Because PowerShell's type names tend to be so long, it's common to refer to them by the last component of the type name. In this case, that would be `ServiceController`, so we can say that `Get-Service` is producing `ServiceController` objects.

Now, let's look at the help for `Stop-Service`.

> **TRY IT NOW** I'm not going to paste the help for `Stop-Service` into this book—go ahead and run `Help Stop-Service` yourself, and follow along with what I'm describing.

There are three variants of `Stop-Service`, or three *parameter sets* (if you forget what a parameter set is, reread chapter 3). Each parameter set seems to provide a different way of specifying the service or services that I want to stop:

- The first parameter set includes a mandatory `-Name` parameter, meaning that I could just specify the service name (or names) I want stopped.
- The second parameter set features a mandatory `-DisplayName` parameter, giving me another way of specifying the service (or services) I want to stop.
- The third parameter set includes an `-InputObject` parameter that accepts values of the type `ServiceController`. That means the `-InputObject` parameter can accept, as its value, the type of object produced by `Get-Service`.

### Above and beyond

Because mastering the help files is such an important PowerShell skill, I want to take a second to focus on some unrelated things in the help file. You're free to skip this brief discussion if you want to, but be sure to come back to it later.

***(continued)***

Of the three parameter sets for `Stop-Service`, the first two have a first parameter that accepts string values. Let's say you ran the command `Stop-Service BITS`. PowerShell needs to decide which of the three parameter sets you've used. You didn't specify a parameter name, so that eliminates the second parameter set, because it doesn't contain any positional parameters (you'll notice that `-DisplayName` is mandatory, and the parameter name itself isn't in square brackets, meaning that if you choose to specify a display name you must type the `-DisplayName` parameter name). Because you provide a string, and not a `ServiceController`, PowerShell knows that you must be intending to use the first parameter set, and so it interprets the value in the first position of your command (`BITS`) as the value for the `-Name` parameter.

There's no way that this cmdlet's designer could have made `-DisplayName` positional (meaning that you wouldn't have to type the `-DisplayName` parameter name, and could just provide a value for it). Doing so would have created two parameter sets that each accepted a string in the first position, and PowerShell wouldn't have been able to tell which one you were trying to use.

These subtle hints from the help file can, once you get used to interpreting them, make it easier to use cmdlets more effectively. Now you know that you can specify the display name of a service you want to start, but if you choose to do so, you'll have to explicitly identify it with the `-DisplayName` parameter name.

Now look at the full help, by running Help `Stop-Service -full`. Scroll down until you get to the help for the `-InputObject` parameter, because I want to look at that in a bit more detail.

Notice that the parameter explanation for `-InputObject` indicates that it isn't required (after all, you could choose to specify a name or display name instead). It's a named parameter, which means that if you choose to use it, you must type the parameter name. And, most importantly for the current discussion, this parameter *accepts pipeline input ByValue.*

When you run a command like `Get-Service -name BITS | Stop-Service`, PowerShell executes the commands in order. `Get-Service` produces those `ServiceController` objects, and then pipes them to `Stop-Service`. PowerShell knows that a cmdlet can only accept input via a parameter, so there's nothing magic associated with piping things from one cmdlet to another. The piped input must be assigned to a parameter of the next cmdlet in order for everything to work, so the shell has to look at all of the parameters for the next cmdlet (`Stop-Service`), and figure out which parameter will accept the objects that have been piped in.

PowerShell starts by looking at the type of object that's being piped. In this case, we know the objects are of the type `ServiceController`, because we used `Gm` to discover that fact. Next, the shell looks to see if any parameters of the next cmdlet are willing to accept that type of object from the pipeline, meaning that the shell looks to see if any parameters accept values of type `ServiceController` *and* are willing to accept that input ByValue from the pipeline. In this instance, PowerShell discovers that the `-InputObject` parameter of `Stop-Service` is willing to accept values of the type `ServiceController`, from the pipeline, ByValue. So the `ServiceController` objects generated by `Get-Service` are passed to the `-InputObject` parameter of `Stop-Service`, which uses those to identify the services we want stopped. So when we say that "a parameter accepts pipeline input ByValue," we're really saying "the parameter will accept incoming objects from the pipeline, so long as those objects match up to the type of value the parameter is designed to accept."

Frankly, I found all that to be a bit confusing the first time someone explained it to me, so let's walk through a few more examples. Start by explaining to yourself why this works:

```
"BITS","MSISCSI" | Start-Service
```

Here's how I walk myself through the explanation:

1. I didn't run a command to put objects in the pipeline. Instead, I manually typed some strings and piped them to the next cmdlet. So the type of object in the pipeline is `String`.
2. Running `Help Start-Service -full`, I see that the `-InputObject` and `-Name` parameters accept pipeline input ByValue. So my strings will attach to one of those two parameters.
3. Of those two, only the `-Name` parameter accepts `String` values, so my strings will attach to `-Name`. Therefore, `Start-Service` will assume that the strings I've piped in are service names, and it will try to start services having those names.

Next, see if you can figure out whether or not this will work (don't actually run the command—just see if you can figure out the explanation):

```
Get-Process -name b* | Stop-Service
```

Again, here's the explanation I would come up with:

1. `Get-Process` is putting something into the pipeline. I would run `Get-Process | Gm` to discover that the objects generated by `Get-Process` are of the type `Process` (technically, `System.Diagnostics.Process`).
2. Looking at the full help for `Stop-Service`, I see that both `-Name` and `-InputObject` are capable of accepting pipeline input by value.
3. Neither `-Name` nor `-InputObject` accept values of the type `Process`, so I would conclude that the preceding command wouldn't work, because `Stop-Service` has no way of accepting the piped-in objects.

That conclusion is correct for as far as we've gotten in this chapter, although we're going to revisit that example later. You'll find that it actually does do something, although it might not be what you want.

Here's one final example for you to try to explain:

```
"conhost" | Stop-Process
```

And here's my explanation:

1. I put an object of type `String` into the pipeline. You could confirm that by running `"conhost" | Gm`.
2. I see that `Stop-Process` has only one parameter that accepts pipeline input ByValue, and that's `-InputObject`.
3. `-InputObject` accepts objects of type `Process`, and not of type `String`, so I conclude that this command will not work.

That turns out to be a correct conclusion.

## 7.3 Parentheses instead of pipelines

The pipeline is only one way to get information into a parameter. You can manually type a simple value like a name or an ID number. But as you've seen in some earlier examples, you can also take the output of one cmdlet and send that output to the parameter of another cmdlet *without* using the pipeline.

For example, take a look at the full help for `Get-Service`, and specifically at the help for its `-computerName` parameter. You'll notice that this parameter accepts pipeline input, but it doesn't do so ByValue. That means I could not pipe in a list of computer names like this:

```
Get-Content c:\names.txt | Get-Service
```

If I ran that command, here's what would happen:

1. `Get-Content` puts objects of type `String` into the pipeline. I can confirm that by running `Get-Content c:\names.txt | Gm` (the assumption is that Names.txt contains a list of computer names, with one name per line).
2. I see that `Get-Service` can accept pipeline input ByValue for its `-InputObject` and `-Name` parameters. Of these, `-Name` accepts values of type `String`.
3. `Get-Service` will accept what is in Names.txt as service names and will try to retrieve those services. It won't treat the names as computer names, which was my intent. So the command will run, but I won't get the results I wanted.

That doesn't mean I can't do what I wanted, but it does mean I can't provide the input to the `-computerName` parameter through the pipeline. Instead, I can use parentheses.

Just like in math class, parentheses change the order in which execution occurs. In the case of math, it's the order in which mathematical expressions are evaluated. In the case of PowerShell, parentheses force the shell to execute certain commands before others.

I already know that the `-computerName` parameter of `Get-Service` can accept multiple string values, because the cmdlet's help lists this:

```
-ComputerName <string[]>
```

Those back-to-back square brackets after `string` are your clue that multiple values are accepted. Therefore, any command that outputs multiple string values can serve as input to the `-computerName` parameter:

```
Get-Service -computerName (Get-Content c:\names.txt)
```

This use of parentheses is a powerful trick for combining commands. You're telling the `-computerName` parameter, "I want you to accept the output of this subcommand, which I have put into parentheses, as your input values." You'll see this again in a more complex example, later in this chapter.

## 7.4 Pipeline input ByPropertyName

If you've been following along and reading the cmdlet help files, you probably noticed a second type of pipeline input. In addition to ByValue, some cmdlets also accept pipeline input ByPropertyName. This second pipeline input mode is a bit more complicated, but it's also very powerful.

The first thing to know is that ByPropertyName mode *only works if ByValue mode does not*. PowerShell always tries to work with pipeline input ByValue if it can, a process sometimes called *pipeline parameter binding ByValue*. When ByValue doesn't turn up any opportunities, PowerShell shifts modes and tries *pipeline parameter binding ByPropertyName*. In this mode, the shell looks at the individual properties of the objects in the pipeline and sees if any of those properties' names happen to match the names of parameters on the next cmdlet. (It only looks at parameters that list "Accept pipeline input: ByPropertyName" in the cmdlets' help.) If it finds matches, the values from those properties are assigned to the parameters that have matching names.

Let's review this example again:

```
Get-Process -name b* | Stop-Service
```

Here's the explanation I would come up with:

1. `Get-Process` is putting something into the pipeline. I would run `Get-Process | Gm` to discover that the objects generated by `Get-Process` are of type `Process` (technically, `System.Diagnostics.Process`).
2. Looking at the full help for `Stop-Service`, I see that both `-Name` and `-InputObject` are capable of accepting pipeline input by value.
3. Neither `-Name` nor `-InputObject` can accept process objects, so ByValue parameter binding ceases. The shell will now try ByPropertyName.
4. The `-Name` parameter is the only one listed as accepting pipeline input ByPropertyName.
5. The objects in the pipeline happen to have a `Name` property, meaning that there is a match between the property name and the `-Name` parameter name.

6. The values of the `Name` properties of the objects in the pipeline—the process names—are assigned to the `-Name` parameter of `Get-Service`. This happens because the property names and the parameter name are the same.
7. `Get-Service` will attempt to stop services, assuming that the process names are the same as service names. That isn't often the case, so it won't stop many services. The BITS service, for example, runs as process name svchost, so attempting to stop the service named "svchost" won't have any effect.

This can be a bit confusing to follow, but it's a powerful technique. Let's look at a real-world example.

> **QUICK REFERENCE** I've included a chart in chapter 28 that can help make sense of the pipeline parameter binding process.

## 7.5 Creating new AD users, fast and easy

If you plan to follow along in this section—and I hope you will—you're going to need to be on a Windows Server 2008 R2 domain controller, or on a Windows 7 computer that has the Remote Server Administration Tools (RSAT) installed, and which is a member of an Active Directory domain that you're allowed to test in and experiment with. We're going to be creating new users, so be sure you're running the shell as a user that has permission to do so (such as a Domain Admin).

Start by loading the ActiveDirectory module into the shell (leave the shell open when you're done with this example, because you'll use the ActiveDirectory module later in this chapter):

```
Import-Module ActiveDirectory
```

> **TRY IT NOW** Make sure you can load this module before proceeding. If you can't, then you don't have the ActiveDirectory module. It only comes with Windows Server 2008 R2, although the RSAT can be used to install it on Windows 7 (but not earlier versions of Windows). Be sure you're either on a test domain controller, or that your Windows 7 computer is a member of a test domain. *You don't want to run the following commands in a production domain!*

Next, use Windows Notepad or Microsoft Office Excel to create a comma-separated values (CSV) file. It's important that you get the column names exactly correct (I'll explain why in a bit). You should also include three to four rows of sample data for new users. Here's my file:

```
samAccountName,Name,Department,City,Title,GivenName,SurName
DonJ,DonJ,IT,Las Vegas,CIO,Don,Jones
GregS,GregS,Janitorial,Denver,Custodian,Greg,Shields
JeffH,JeffH,IT,Syracuse,Technician,Jeffery,Hicks
ChrisG,ChrisG,Finance,Las Vegas,Accountant,Christopher,Gannon
```

Those are all good friends, by the way—Greg always gets to be the janitor in these little examples.

Save the file as C:\Users.csv, and if you're using Notepad, don't forget to surround the entire file path and name with quotation marks, so that Notepad won't add the .txt filename extension. This example won't work if the file is named C:\Users.csv.txt (and remember that Explorer will hide the .txt filename extension by default).

> **TIP** In PowerShell, run `notepad c:\users.csv`. If the file doesn't exist, Notepad will offer to create it, and it won't tack on the .txt filename extension.

Now, look at the full help for the `New-ADUser` cmdlet. Specifically, pay attention to which parameters accept pipeline input ByPropertyName and which ones accept pipeline input ByValue. You should come to the conclusion that nearly every parameter will work ByPropertyName, and not a single one of them supports ByValue. Remember that—we'll come back to it in a minute.

Now, use `Import-CSV` to load that newly created CSV file into the shell. Just let it display its information on the screen:

```
PS C:\> import-csv users.csv

samAccountName : DonJ
Name           : DonJ
Department     : IT
City           : Las Vegas
Title          : CIO
GivenName      : Don
SurName        : Jones

samAccountName : GregS
Name           : GregS
Department     : Janitorial
City           : Denver
Title          : Custodian
GivenName      : Greg
SurName        : Shields
```

Your output should contain all of the users that you entered into the CSV file. Here's what happened: `Import-CSV` translates the CSV file, breaking out each column and row for you. It constructs an object for each row in the file, and each column in the file becomes a property of those objects. I had four users in the file, so `Import-CSV` generated four objects. Each object has properties named after the columns in the CSV file's header row.

Piping the output to `Gm` confirms this analysis:

```
PS C:\> import-csv users.csv | gm

   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()
City             NoteProperty    System.String City=Las Vegas
Department       NoteProperty    System.String Department=IT
GivenName        NoteProperty    System.String GivenName=Don
Name             NoteProperty    System.String Name=DonJ
samAccountName   NoteProperty    System.String samAccountName=DonJ
SurName          NoteProperty    System.String SurName=Jones
Title            NoteProperty    System.String Title=CIO
```

You can see that the object is of type `PSCustomObject`, and it has (in addition to a few system-generated methods) a property for each column in my CSV file: City, Department, GivenName, and so forth.

Now for the magic: go back to the help for `New-ADUser`. Considering each of the seven properties in my CSV file, can you find parameters of `New-ADUser` that match those property names and that accept pipeline input ByPropertyName?

You should be able to identify all seven properties as matching parameter names of `New-ADUser`, and all seven of those parameters accept pipeline input ByPropertyName. That means I can run the following command and it should work:

```
Import-CSV c:\names.csv | New-ADUser
```

It might not work if those users already exist (if you run the command a second time, for example), but in a fresh test domain it should work fine. The users will be created in the default Users container. If you wanted them to be created elsewhere, say in a Sales OU, you could have run this instead:

```
Import-CSV c:\names.csv | New-ADUser -path "OU=Sales,dc=Company,dc=pri"
```

It's perfectly acceptable to specify some parameters manually, with others being set through the pipeline.

Here are some other notes:

- You can include as many columns as you want in that CSV file, provided each column name exactly matches a parameter of `New-ADUser`.
- You can have extraneous columns in the CSV file. If they don't match a parameter of `New-ADUser`, they'll be ignored.
- You can manually specify any parameter you like; whatever value you provide will override anything coming in from the pipeline for that parameter, and your value will be effective for every new user that's created.

As you can see, this is an extremely powerful technique! You've created any number of new users with little effort. Assuming you can get someone else to hand you that CSV file—say, the Human Resources department—then a single command line turns that data into new user accounts!

This really demonstrates PowerShell's flexibility: with very little typing, you've automated something that could have taken hours, depending on the number of users you had to manually create. With PowerShell, one user or one hundred users can be created in a couple of seconds.

## 7.6 When things don't line up: custom properties

Speaking of the Human Resources department, what are the odds that they'll give you a properly formatted CSV file every time? Remember that ByPropertyName only works if the CSV column names exactly match those parameter names—will HR get it right every time? Possibly not. Possibly, you'll get a CSV that looks like this instead:

```
LoginName,Department,City,Title,FirstName,LastName
DonJ,IT,Las Vegas,CIO,Don,Jones
GregS,Janitorial,Denver,Custodian,Greg,Shields
JeffH,IT,Syracuse,Technician,Jeffery,Hicks
ChrisG,Finance,Las Vegas,Accountant,Christopher,Gannon
```

Obviously, you could rename columns yourself. You'd have to add a column to each line, though, because AD needs both a `samAccountName` property and a `Name` property, and those usually match. This file only has one column. But there's no need to do any of that manually: let's make PowerShell do the hard work for us.

We're going to use a cmdlet that you've seen before, `Select-Object`. But rather than just selecting existing properties to use, we're going to have it create brand-new properties for us as well. I want to acknowledge in advance that the syntax for doing this is really, really ugly, but if you can memorize it (or jot it down into a notepad for future reference), you'll find a number of places where it can be used.

For each new property that we create, we need to provide a property name, or label, and a value for the property, which is specified in an expression. "Label" is generally abbreviated as a lowercase "L," and the expression as a lowercase "e." We're happy with the Department, City, and Title columns in the CSV file, but we need to create Surname, GivenName, samAccountName, and Name. The latter two need to both pull from the LoginName column we've been given in the CSV file. Here we go:

> **TRY IT NOW** If you create a CSV file like the one I've listed above, and name it C:\users2.csv, you'll be able to follow along with this command. Also note that I'm going to break this command onto several lines, both to make it fit in the book and for easier reading, but this should all be typed as a single, long command on a single line. Notice that the user names are the same as the previous example, so if you've already created these users, delete them from the domain before proceeding.

```
Import-CSV c:\users2.csv |
➥ Select-Object *,@{l='samAccountName';e={$_.LoginName}},
                ➥ @{l='Name';e={$_.LoginName}},
                ➥ @{l='GivenName';e={$_.FirstName}},
                ➥ @{l='Surname';e={$_.LastName}}
```

> **TRY IT NOW** Be cautious when typing—those are lowercase "L" letters, not the number 1.

Notice with `Select-Object` that I started by specifying `*` in the property list. That will select all properties that the objects already have, meaning all of the CSV file's columns will show up in the output. I then specified four new columns by creating four specially formatted constructs called *hashtables*. The Select-Object cmdlet is specifically designed to accept this kind of construct. Each hashtable consists of two elements, and each element has both a *key* and a *value*. For the L, or Label key, the value is the name of the new property I want to add. For the E, or Expression key, the value is what's called a *script block*. Enclosed in curly braces, like { }, this script block tells PowerShell how to create the value for that property. PowerShell would also permit the use of N or Name instead of L or Label.

Inside that expression, PowerShell lets us use a special placeholder: $_ (often pronounced as *dollar underscore* or *dollar underbar*). When the command runs, PowerShell will fill in this placeholder with the objects that were piped into Select-Object. Therefore, $_ will represent the rows from the CSV file. After the underscore, I've typed a period, which tells the shell that I don't want to refer to the entire row from the CSV file, but rather to access a single property (or column). This is how I created a new property called samAccountName and had it pull over the value from the CSV file's LoginName column.

Again, I realize that there's a lot of punctuation flying around in that example, but this is honestly one of the trickiest, most punctuation-intensive things you'll see in this book. If you can spend enough time looking at this example to become familiar with it, and be very careful when you're typing, this is a pattern that you'll be able to adapt and re-use in your own projects without too much pain.

The output of the previous command should be a revised list of objects, listing both the properties from the CSV file as well as the four additional properties I specified. If you're satisfied with that output, the only remaining step is to send it to New-ADUser:

```
Import-CSV c:\users2.csv |
➥ Select-Object *,{l='samAccountName';e={$_.LoginName}},
                  ➥ {l='Name';e={$_.LoginName}},
                  ➥ {l='GivenName';e={$_.FirstName}},
                  ➥ {l='Surname';e={$_.LastName}} |
➥ New-ADUser
```

Notice that I didn't include New-ADUser the first time, which let me preview the output of Select-Object and make sure that it was what I wanted. I could then go back and tweak it a bit, if I wanted, and not pipe everything to New-ADUser until I was completely satisfied with the results.

## 7.7 Extracting the value from a single property

Earlier in this chapter, I showed you an example of using parentheses to execute Get-Content, feeding its output to the parameter of another cmdlet:

```
Get-Service -computerName (Get-Content names.txt)
```

Rather than getting your computer names from a static text file, you might well want to query them from Active Directory. With the ActiveDirectory module (which hopefully you still have loaded from the previous examples in this chapter), you could query all of your domain controllers:

```
get-adcomputer -filter * -searchbase "ou=domain controllers,
➥ dc=company,dc=pri"
```

Could you use the same parentheses trick? For example, would this work?

```
Get-Service -computerName (Get-ADComputer -filter *
➥ -searchBase "ou=domain controllers,dc=company,dc=pri")
```

Sadly, it won't. Look at the help for `Get-Service`, and you'll see that the `-computerName` parameter expects `String` values.

Run this instead:

```
get-adcomputer -filter * -searchbase "ou=domain controllers,
➥ dc=company,dc=pri" | gm
```

`Get-Member` reveals that `Get-ADComputer` is producing objects of the type `ADComputer`. Those aren't `String` objects, so `-computerName` won't know what to do with them. The `ADComputer` objects do have a `Name` property, however. What we need to do is somehow extract just the values of the objects' `Name` properties, and feed those values, which are computer names, to the `-ComputerName` parameter.

Once again, the `Select-Object` cmdlet can rescue us. It includes an `-expandProperty` parameter, which accepts a property name. It will take that property and extract its values, and return just those values as the output of `Select-Object`. Try this:

```
get-adcomputer -filter * -searchbase "ou=domain controllers,
➥ dc=company,dc=pri" | Select-Object -expand name
```

You should get a simple list of computer names. Those can be fed to the `-computerName` parameter of `Get-Service` (or any other cmdlet that has a `-computerName` parameter):

```
Get-Service -computerName (get-adcomputer -filter *
➥ -searchbase "ou=domain controllers,dc=company,dc=pri" |
➥ Select-Object -expand name)
```

Again, this is a cool trick that makes it possible to combine an even wider variety of commands with each other, saving you typing and making PowerShell do more of the work.

## 7.8 Lab

Once again, we've covered a lot of important concepts in a short amount of time. The best way to cement your new knowledge is to put it to immediate use. I recommend doing the following tasks in order, because they build on each other to help remind you of what you've learned and to help you find practical ways to use that knowledge. Complete these tasks:

1. Would the following command work to retrieve a list of installed hotfixes from all domain controllers in the specified domain? Why or why not? Write out an explanation, similar to the ones I provided earlier in this chapter.

   ```
   Get-Hotfix -computerName (get-adcomputer -filter *
   ➥ -searchbase "ou=domain controllers,dc=company,dc=pri" |
   ➥ Select-Object -expand name)
   ```

2 Would this alternative command work to retrieve the list of hotfixes from the same computers? Why or why not? Write out an explanation, similar to the ones I provided earlier in this chapter.

```
get-adcomputer -filter *
➥ -searchbase "ou=domain controllers,dc=company,dc=pri" |
➥ Get-HotFix
```

3 Would this third version of the command work to retrieve the list of hotfixes from the domain controllers? Why or why not? Write out an explanation, similar to the ones I provided earlier in this chapter.

```
get-adcomputer -filter *
➥ -searchbase "ou=domain controllers,dc=company,dc=pri" |
➥ Select-Object @{l='computername';e={$_.name}} |
➥ Get-Hotfix
```

4 Write a command that uses pipeline parameter binding to retrieve a list of running processes from every computer in an AD domain. Don't use parentheses.

5 Write a command that retrieves a list of installed services from every computer in an AD domain. Don't use pipeline input; instead use a parenthetical command (a command in parentheses).

6 Sometimes Microsoft forgets to add pipeline parameter binding to a cmdlet. For example, would the following command work to retrieve information from every domain controller in the domain? Write out an explanation, similar to the ones I provided earlier in this chapter.

```
get-adcomputer -filter *
➥ -searchbase "ou=domain controllers,dc=company,dc=pri" |
➥ Select-Object @{l='computername';e={$_.name}} |
➥ Get-WmiObject -class Win32_BIOS
```

# 8 Formatting—and why it's done on the right

Let's quickly review: you know that PowerShell cmdlets produce objects, and that those objects often contain more properties than PowerShell shows by default. You know how to use `Gm` to get a list of all of an object's properties, and you know how to use `Select-Object` to specify the properties you want to see. Up to now, you've relied on PowerShell's default configuration and rules to determine how the final output will appear on the screen (or in a file, or in hardcopy form). In this chapter, you'll learn to override those defaults and create your own formatting for your commands' output.

## 8.1 Formatting: making what you see prettier

I don't want to give you the impression that PowerShell is a full-fledged management reporting tool, because it isn't. But PowerShell has good capabilities for collecting information about computers, and, with the right output, you can certainly produce reports using that information. The trick, of course, is getting the right output, and that's what formatting is all about.

On the surface, PowerShell's formatting system can seem pretty easy to use—and for the most part that's true. But the formatting system also contains some of the trickiest "gotchas" in the entire shell, so I want to make sure you really understand how it works and why it does what it does. I'm not just going to show you a few new commands here, but rather explain how the entire system works, how you can interact with it, and what limitations you might run into.

## 8.2 About the default formatting

Run our old friend `Get-Process` again, and pay special attention to the column headers. Notice that they don't exactly match the property names. Instead, they each have a specific width, alignment, and so forth. All that configuration stuff has to come from someplace, right? You'll find it in one of the .format.ps1xml files that install with PowerShell. Specifically, formatting directions for process objects are in DotNetTypes .format.ps1xml.

> **TRY IT NOW** You'll definitely want to have PowerShell open so that you can follow along with what I'm about to show you. This will really help you understand what the formatting system is up to under the hood.

Start by changing to the PowerShell installation folder and opening DotNetTypes .format.ps1xml. Be careful not to save any changes to this file! It's digitally signed, and any changes that you save—even a single carriage return or space added to the file—will break the signature and prevent PowerShell from using the file.

```
PS C:\>cd $pshome
PS C:\>notepad dotnettypes.format.ps1ml
```

Next, find out the exact type of object returned by `Get-Process`:

```
PS C:\>get-process | gm
```

Now follow these steps:

1. Copy and paste the complete type name, `System.Diagnostics.Process`, to the clipboard. Do to so, use your cursor to highlight the type name, and press Return to copy it to the clipboard.
2. Switch over to Notepad and press Ctrl-F to open the Find window.
3. In the Find window, paste in the type name you copied to the clipboard. Click Find Next.
4. The first thing you find will probably be a `Process-Module` object, not a `Process` object, so click Find Next again and again until you locate `System.Diagnostics.Process` in the file. Figure 8.1 shows what you should have found.

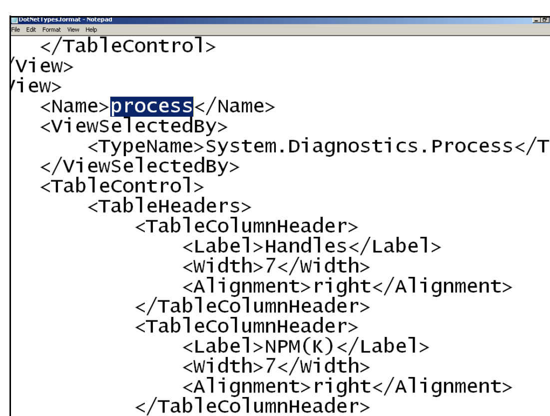



**Figure 8.1 Locating the Process view in Windows Notepad**

What you're now looking at in Notepad is the set of directions that govern how a process is displayed by default. Scroll down a bit, and you'll see the definition for a *table view*, which you should expect because you already know that processes display in a multicolumn table. You'll see the familiar column names, and if you scroll down a bit more you'll see where the file specifies which property will display in each column. You'll see definitions for column widths and alignments too. When you're done browsing, close Notepad, being careful not to save any changes that you may have accidentally made to the file, and go back to PowerShell.

When you run `Get-Process`, here's what happens in the shell:

1 The cmdlet places objects of the type `System.Diagnostics.Process` into the pipeline.
2 At the end of the pipeline is an invisible cmdlet called `Out-Default`. It's always there, and its job is to pick up whatever objects are in the pipeline after all of your commands have run.
3 `Out-Default` passes the objects to `Out-Host`, because the PowerShell console is designed to use the screen (called the *host*) as its default form of output. In theory, someone could write a shell that uses files or printers as the default output instead, but nobody has that I know of.
4 Most of the `Out-` cmdlets are incapable of working with normal objects. Instead, they're designed to work with special formatting instructions. So when `Out-Host` sees that it has been handed normal objects, it passes them to the formatting system.
5 The formatting system looks at the type of the object and follows an internal set of formatting rules (we'll cover those in a moment). It uses those rules to produce formatting instructions, which are passed back to `Out-Host`.
6 Once `Out-Host` sees that it has formatting instructions, it follows those instructions to construct the onscreen display.

All of this happens whenever you manually specify an `Out-` cmdlet, too. For example, run `Get-Process | Out-File procs.txt`, and `Out-File` will see that you've sent it some normal objects. It will pass those to the formatting system, which creates formatting instructions and passes them back to `Out-File`. `Out-File` then constructs the text file based on those instructions. So the formatting system becomes involved anytime objects need to be converted into human-readable textual output.

What rules does the formatting system follow in step 5, above? For the first formatting rule, the system looks to see if the type of object it's dealing with has a predefined view. That's what you saw in DotNetTypes.format.ps1xml: a predefined view for a `Process` object. There are a few other .format.ps1xml files installed with PowerShell, and they're all loaded by default when the shell starts. You can create your own predefined views as well, although doing so is beyond the scope of this book.

The formatting system looks for predefined views that specifically target the object type it's dealing with—meaning that in this case it's looking for the view that handles `System.Diagnostics.Process` objects.

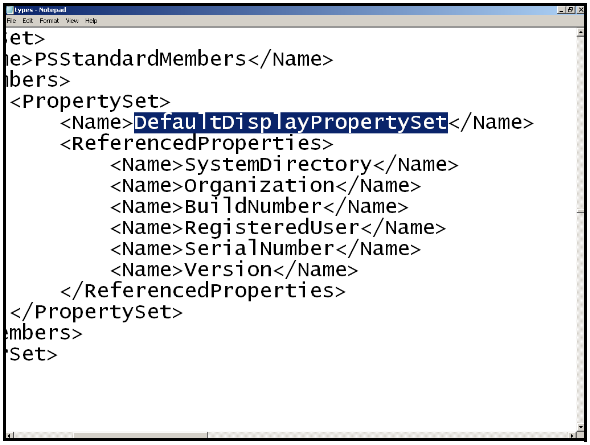



**Figure 8.2 Locating a `DefaultDisplayPropertySet` in Notepad**

What if there is no predefined view? For example, try running this:

```
Get-WmiObject Win32_OperatingSystem | Gm
```

Grab that object's type name (or at least the "Win32_OperatingSystem" part), and try to find it in one of the .format.ps1xml files. I'll save you some time by telling you that you won't find it.

This is where the formatting system takes its next step, or what I call the second formatting rule: it looks to see if anyone has declared a default display property set for that type of object. You'll find those in a different configuration file, Types.ps1xml. Go ahead and open it in Notepad now (again, be careful not to save any changes to this file) and use the Find function to locate Win32_OperatingSystem. Once you do, scroll down a bit and you'll see `DefaultDisplayPropertySet`. It's shown in figure 8.2. Make a note of the six properties listed there.

Now, go back to PowerShell and run this:

```
Get-WmiObject Win32_OperatingSystem
```

Do the results look familiar? They should: the properties you see are there solely because they're listed as defaults in Types.ps1xml. If the formatting system finds a default display property set, it will use that set of properties for its next decision. If it doesn't find one, the next decision will consider all of the object's properties.

That next decision—the third formatting rule—is about what kind of output to create. If the formatting system will display four or fewer properties, it will use a table. If there are five or more properties, it will use a list. That's why the Win32_OperatingSystem object wasn't displayed as a table: there were six properties, triggering a list. The theory is that more than four properties might not fit well into an ad hoc table without truncating information.

Now you know how the default formatting works. You also know that most `Out-` cmdlets will automatically trigger the formatting system, so that they can get the formatting instructions they need. Next let's look at how we can control that formatting system ourselves, and override the defaults.

## 8.3 Formatting tables

There are four formatting cmdlets in PowerShell, and we'll work with the three that provide the most day-to-day formatting capability (the fourth is briefly discussed in an "Above and beyond" section near the end of this chapter). First up is `Format-Table`, which has an alias, `Ft`.

If you read the help file for `Format-Table`, you'll notice that it has a number of parameters. These are some of the most useful ones, along with examples of how to use them:

- `-autoSize`—Normally, PowerShell tries to make a table fill the width of your window (the exception is when a predefined view, like the one for processes, defines column widths). That means a table with relatively few columns will have a lot of space in between those columns, which isn't always attractive. By adding the `-autosize` parameter, you force the shell to try to size each column to hold its contents, and no more. This makes the table a bit "tighter" in appearance, although it will take a bit of extra time for the shell to start producing output. That's because it has to examine every object that will be formatted to find the longest values for each column. Here's an example:

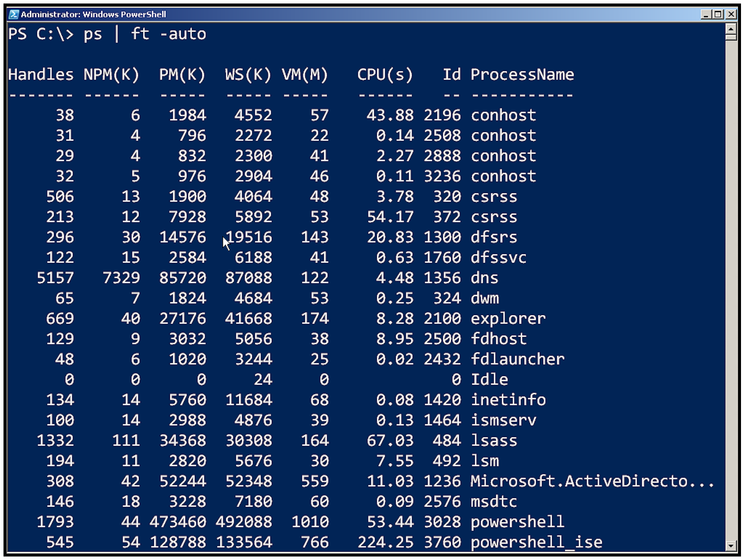
  ```
  Get-WmiObject Win32_BIOS | Format-Table -autoSize
  ```

- `-property`—This parameter accepts a comma-separated list of properties that should be included in the table. These properties aren't case-sensitive, but the shell will use whatever you type as the column headers, so you can get nicer-looking output by properly casing the property names ("CPU" instead of "cpu," for example). This parameter accepts wildcards, meaning you can specify `*` to include all properties in the table, or something like `c*` to include all properties starting with *c*. Note that the shell will still only display the properties it can fit in the table, so not every property you specify may display. This parameter is positional, so you don't have to type the parameter name, provided the property list is in the first position. Try these examples (the last one is shown in figure 8.3):

  ```
  Get-Process | Format-Table -property *
  Get-Process | Format-Table -property ID,Name,Responding -autoSize
  Get-Process | Format-Table * -autoSize
  ```

- `-groupBy`—This parameter generates a new set of column headers each time the specified property value changes. This only works well when you have first sorted the objects on that same property. An example is the best way to see how this works:

  ```
  Get-Service | Sort-Object Status | Format-Table -groupBy Status
  ```



```
Administrator: Windows PowerShell
PS C:\> ps | ft -auto

Handles NPM(K)  PM(K)  WS(K) VM(M)   CPU(s)   Id ProcessName
------- ------  -----  ----- -----   ------   -- -----------
     38      6   1984   4552    57    43.88 2196 conhost
     31      4    796   2272    22     0.14 2508 conhost
     29      4    832   2300    41     2.27 2888 conhost
     32      5    976   2904    46     0.11 3236 conhost
    506     13   1900   4064    48     3.78  320 csrss
    213     12   7928   5892    53    54.17  372 csrss
    296     30  14576  19516   143    20.83 1300 dfsrs
    122     15   2584   6188    41     0.63 1760 dfssvc
   5157   7329  85720  87088   122     4.48 1356 dns
     65      7   1824   4684    53     0.25  324 dwm
    669     40  27176  41668   174     8.28 2100 explorer
    129      9   3032   5056    38     8.95 2500 fdhost
     48      6   1020   3244    25     0.02 2432 fdlauncher
      0      0      0     24     0             0 Idle
    134     14   5760  11684    68     0.08 1420 inetinfo
    100     14   2988   4876    39     0.13 1464 ismserv
   1332    111  34368  30308   164    67.03  484 lsass
    194     11   2820   5676    30     7.55  492 lsm
    308     42  52244  52348   559    11.03 1236 Microsoft.ActiveDirecto...
    146     18   3228   7180    60     0.09 2576 msdtc
   1793     44 473460 492088  1010    53.44 3028 powershell
    545     54 128788 133564   766   224.25 3760 powershell_ise
```

**Figure 8.3 Creating an auto-sized table of processes**

- `-wrap`—If the shell has to truncate information in a column, it will end that column with ellipses (`...`) to visually indicate that information was suppressed. This parameter enables the shell to wrap information, which will make the table longer, but will preserve all of the information you wanted to display. Here's an example:

  ```
  Get-Service | Format-Table Name,Status,DisplayName -autoSize -wrap
  ```

**TRY IT NOW** You should run through all of these examples in the shell, and feel free to mix and match these techniques. Experiment a bit to see what works, and what sort of output you can create.

## 8.4 Formatting lists

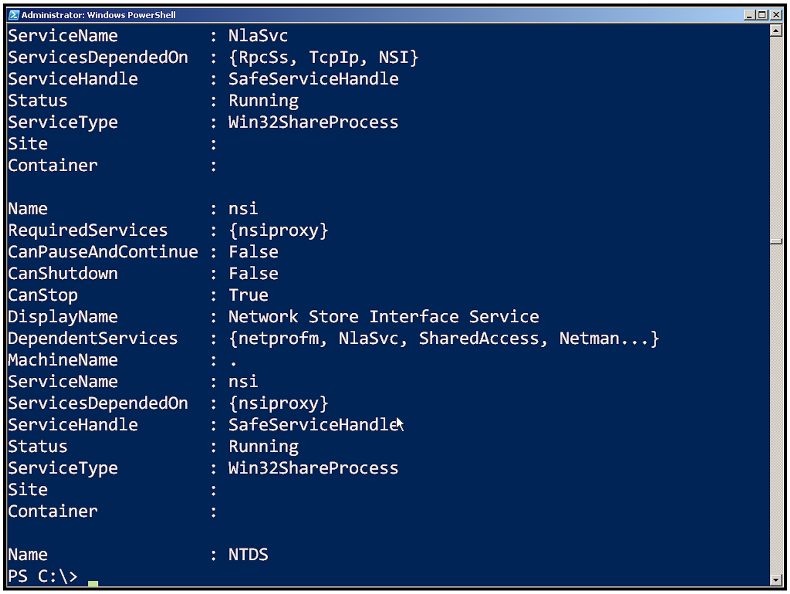
Sometimes you need to display more information than will fit horizontally in a table, which can make a list useful. `Format-List` is the cmdlet you'll turn to, or you can use its alias, `Fl`.

This cmdlet supports some of the same parameters as `Format-Table`, including `-property`. In fact, `Fl` is another way of displaying the properties of an object. Unlike `Gm`, `Fl` will also display the values for those properties, so that you can see what kind of information each property contains:

```
Get-Service | Fl *
```

Figure 8.4 shows an example of the output. I often use `Fl` as an alternative way of discovering the properties of an object.

**TRY IT NOW** Read the help for `Format-List`, and try experimenting with its different parameters.



```
ServiceName         : NlaSvc
ServicesDependedOn  : {RpcSs, TcpIp, NSI}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
Site                :
Container           :

Name                : nsi
RequiredServices    : {nsiproxy}
CanPauseAndContinue : False
CanShutdown         : False
CanStop             : True
DisplayName         : Network Store Interface Service
DependentServices   : {netprofm, NlaSvc, SharedAccess, Netman...}
MachineName         : .
ServiceName         : nsi
ServicesDependedOn  : {nsiproxy}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
Site                :
Container           :

Name                : NTDS
PS C:\>
```

**Figure 8.4 Reviewing services displayed in list form**

## 8.5 Formatting wide

The last cmdlet, `Format-Wide` (or its alias, `Fw`), displays a wide list. It's able to display only the values of a single property, so its `-property` parameter accepts only one property name, not a list, and it can't accept wildcards.

By default, `Format-Wide` will look for an object's `Name` property, because `Name` is a commonly used property and usually contains useful information. The display will generally default to two columns, but a `-columns` parameter can be used to specify more columns:

```
Get-Process | Format-Wide name -col 4
```

Figure 8.5 shows an example of what you should see.

> **TRY IT NOW** Read the help for `Format-Wide`, and try experimenting with its different parameters.

## 8.6 Custom columns and list entries

Flip back to the previous chapter, and review the section entitled, "When things don't line up: custom properties." In that section, I showed you how to use a hashtable construct to add custom properties to an object. Both `Format-Table` and `Format-List` can use those same constructs to create custom table columns or custom list entries.

You might do this to provide a column header that's different from the property name being displayed:

```
Get-Service |
➥ Format-Table @{l='ServiceName';e={$_.Name}},Status,DisplayName
```

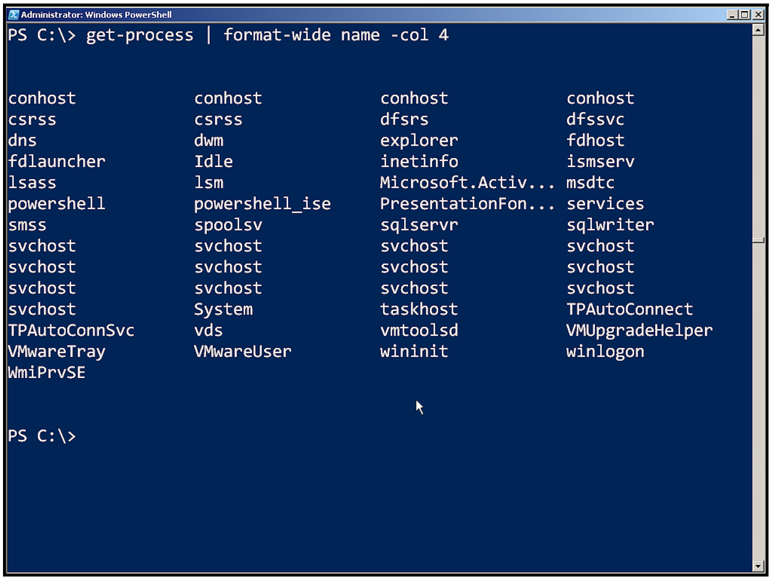



**Figure 8.5 Displaying process names in a wide list**

Or, you might put a more complex mathematical expression in place:

```
Get-Process |
➥ Format-Table Name,
➥ @{l='VM(MB)';e={$_.VM / 1MB -as [int]}} -autosize
```

Figure 8.6 shows the output of the preceding command. I admit, I'm cheating here a little bit by throwing in a bunch of stuff that we haven't talked about yet.

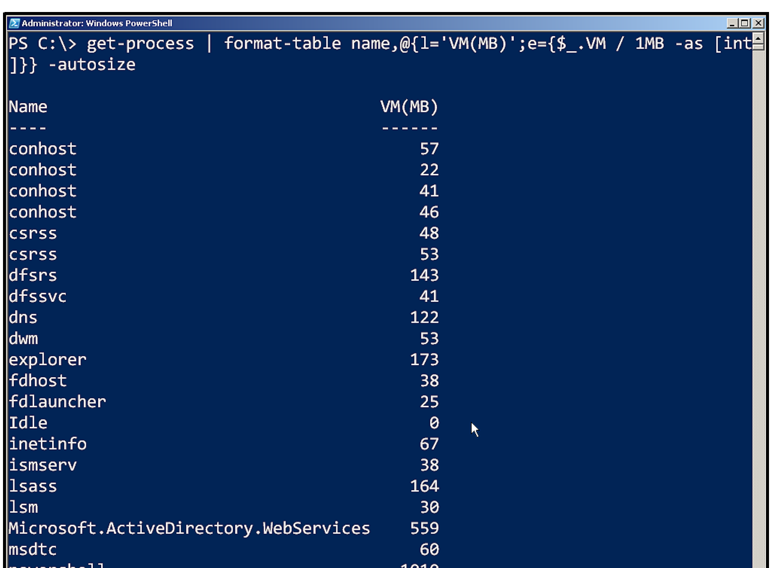



**Figure 8.6 Creating a custom, calculated table column**

We might as well talk about it now!

- Obviously, I'm starting with `Get-Process`, a cmdlet you're more than familiar with by now. If you run `Get-Process | Fl *`, you'll see that the `VM` property is in bytes—although that's not how the default table view displays it.
- I'm telling `Format-Table` to start with the process's `Name` property.
- Next, I'm creating a custom column that will be labeled `VM(MB)`. The value, or expression, for that column takes the object's normal `VM` property and divides it by `1MB`. The slash is PowerShell's division operator, and PowerShell recognizes the shortcuts `KB`, `MB`, `GB`, `TB`, and `PB` as denoting kilobyte, megabyte, gigabyte, terabyte, and petabyte respectively.
- The result of that division operation will have a decimal component that I don't want to see. The `-as` operator enables me to change the data type of that result from a floating-point value to, in this case, an integer value (specified by `[int]`). The shell will round up or down, as appropriate, when making that conversion. The result is a whole number with no fractional component.

I wanted to show you this little division-and-changing trick because it can be really useful in creating nicer-looking output. We won't spend much more time in this book on these operations (although I will tell you that `*` is used for multiplication, and as you might expect `+` and `-` are for addition and subtraction).

> **Above and beyond**
>
> I'd like you to try repeating the previous example, but this time don't type it all on one line. Type it exactly as it's shown here in the book, on three lines total. You'll notice after typing the first line, which ends in a pipe character, that PowerShell changes its prompt. That's because you ended the shell in a pipe, and the shell knows that there are more commands coming. It will enter this same "waiting for you to finish" mode if you hit Return without properly closing all curly braces, quotation marks, and parentheses.
>
> If you didn't mean to enter that extended-typing mode, hit Ctrl-C to abort, and start over. In this case, you could type the second line of text and hit Return, and then type the third line and hit Return. In this mode, you'll have to hit Return one last time, on a blank line, to tell the shell you're done. When you do so, it will execute the command as if it had been typed on a single, continuous line.

## 8.7 Going out: to a file, a printer, or the host

Once something is formatted, you have to decide where it will go.

If a command line ends in a `Format-` cmdlet, the formatting instructions created by the `Format-` cmdlet will go to `Out-Default`, which forwards them to `Out-Host`, which displays them on the screen:

```
Get-Service | Format-Wide
```

You could also manually pipe the formatting instructions to `Out-Host`, which would accomplish exactly the same thing:

```
Get-Service | Format-Wide | Out-Host
```

Alternatively, you can pipe formatting instructions to either `Out-File` or `Out-Printer` to direct formatted output to a file or to hardcopy. As you'll read later, in "Common points of confusion," only one of those three `Out-` cmdlets should ever follow a `Format-` cmdlet on the command line.

Keep in mind that both `Out-Printer` and `Out-File` default to a specific character width for their output, which means a hardcopy or a text file might look different from what would display on the screen. The cmdlets have a `-width` parameter that enables you to change the output width, if desired, to accommodate wider tables.

## 8.8 Another out: GridViews

You've seen `Out-GridView` in previous chapters, and I mention it here because it's another useful form of output. Note that this isn't technically formatting; in fact, `Out-GridView` entirely bypasses the formatting subsystem. No `Format-` cmdlets are called, no formatting instructions are produced, and no text output is displayed in the console window. `Out-GridView` can't receive the output of a `Format-` cmdlet—it can only receive the regular objects output by other cmdlets.

Figure 8.7 shows what the grid view looks like.

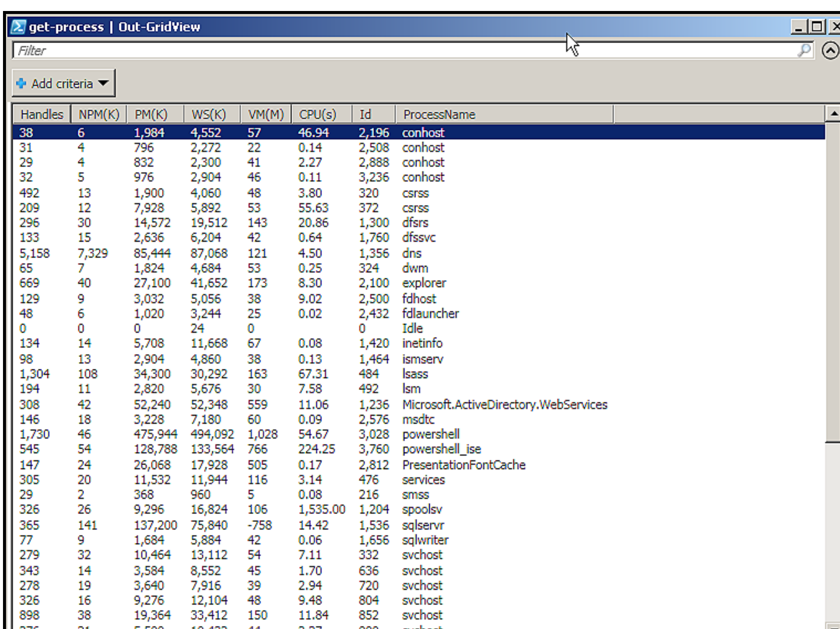



**Figure 8.7 The results of the `Out-GridView` cmdlet**

## 8.9 Common points of confusion

As I mentioned at the start of this chapter, the formatting system has most of the gotchas that trip up PowerShell newcomers. There are two main things that my classroom students tend to run across, so I'll try to help you avoid them.

### 8.9.1 Always format right

It's incredibly important that you remember one rule from this chapter: *format right*. In other words, your `Format-` cmdlet should be the last thing on the command line, with `Out-File` or `Out-Printer` as the only real exceptions. The reason for this rule is that the `Format-` cmdlets produce formatting instructions, and only an `Out-` cmdlet can properly consume those instructions. If a `Format-` cmdlet is last on the command line, the instructions will go to `Out-Default` (which is always at the end of the pipeline), which will forward them to `Out-Host`, which is happy to work with formatting instructions.

Try running this command to illustrate the need for this rule:

```
Get-Service | Format-Table | Gm
```

You'll notice, as shown in figure 8.8, that `Gm` isn't displaying information about your service objects, because the `Format-Table` cmdlet doesn't output service objects. It consumes the service objects you piped in, and it outputs formatting instructions—which is what `Gm` sees and reports on.

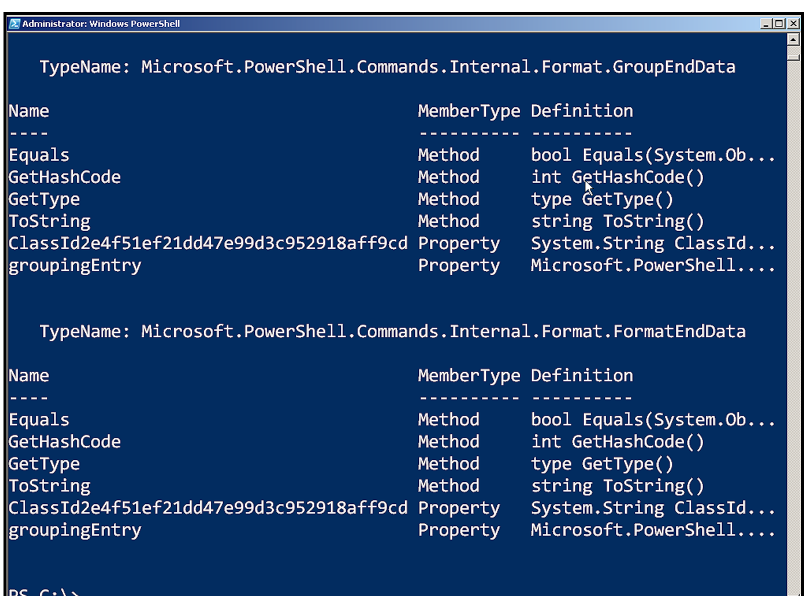



**Figure 8.8 Formatting cmdlets produce special formatting instructions, which aren't meaningful to humans.**

Now try this:

```
Get-Service | Select Name,DisplayName,Status | Format-Table |
➥ ConvertTo-HTML | Out-File services.html
```

Go ahead and open Services.html in Internet Explorer, and you'll see some pretty crazy results. You didn't pipe service objects to `ConvertTo-HTML`; you piped formatting instructions, so that's what got converted to HTML. This illustrates why a `Format-` cmdlet, if you use one, either has to be the last thing on the command line, or has to be second-to-last with the last cmdlet being `Out-File` or `Out-Printer`.

Also know that `Out-GridView` is unusual (for an `Out-` cmdlet, at least) in that it *won't* accept formatting instructions and *will* only accept normal objects. Try these two commands to see the difference:

```
PS C:\>Get-Process | Out-GridView
PS C:\>Get-Process | Format-Table | Out-GridView
```

That's why I explicitly mentioned `Out-File` and `Out-Printer` as the only cmdlets that should follow a `Format-` cmdlet (technically, `Out-Host` can also follow a `Format-` cmdlet, but there's no need because ending the command line with the `Format-` cmdlet will get the output to `Out-Host` anyway).

### 8.9.2 One type of object at a time, please

The next thing to avoid is putting multiple kinds of objects into the pipeline. The formatting system looks at the first object in the pipeline and uses the type of that object to determine what formatting to produce. If the pipeline contains two or more kinds of objects, the output won't always be complete or useful.

For example, run this:

```
Get-Process; Get-Service
```

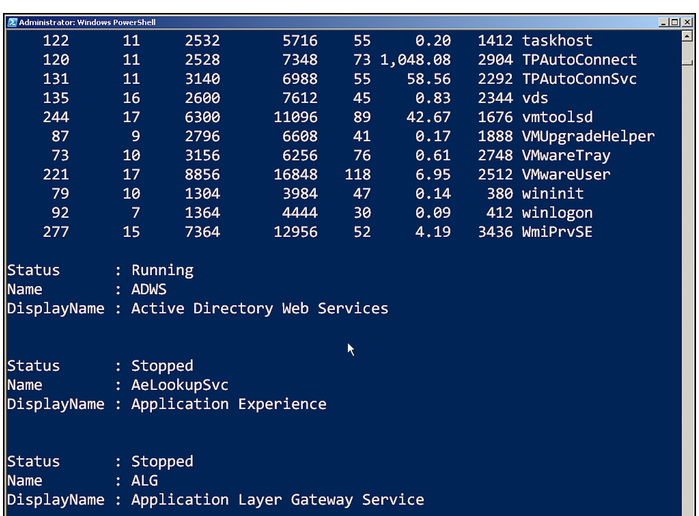



**Figure 8.9 Putting two types of objects into the pipeline at once can confuse PowerShell's formatting system.**

That semicolon allows me to put two commands onto a single command line, without piping the output of the first cmdlet into the second one. In other words, both cmdlets will run independently, but they will put their output into the same pipeline. As you'll see if you try this, or look at figure 8.9, the output starts out fine, displaying process objects. But the output breaks down when it's time to display the service objects. Rather than producing the table you're used to, PowerShell reverts to a list. The formatting system simply isn't designed to take multiple kinds of objects and make the results look as attractive as possible.

> **Above and beyond**
>
> Technically, the formatting system *can* handle multiple types of objects—if you tell it how. Run `Dir | Gm` and you'll notice that the pipeline contains both `DirectoryInfo` and `FileInfo` objects (`Gm` has no problem working with pipelines that contain multiple kinds of objects and will display member information for all of them). When you run `Dir` by itself, the output is perfectly legible. That's because Microsoft provides a predefined custom formatting view for `DirectoryInfo` and `FileInfo` objects, and that view is handled by the `Format-Custom` cmdlet.
>
> `Format-Custom` is mainly used to display different predefined custom views. You could technically create your own predefined custom views, but the necessary XML syntax is complicated and isn't publicly documented at this time. So custom views are pretty much limited to what Microsoft provides.
>
> Microsoft's custom views do get a lot of usage, though. PowerShell's help information is stored as objects, for example, and the formatted help files you see on the screen are the result of feeding those objects into a custom view.

What if you want to combine information drawn from two (or more) different places into a single form of output? You absolutely can, and you can do so in a way that the formatting system can deal with very nicely. But you have a lot more to learn before that—I'll get to it in chapter 19.

## 8.10 Lab

See if you can complete the following tasks:

1. Display a table of processes that includes only the process names, IDs, and whether or not they're responding to Windows (the `Responding` property has that information). Have the table take up as little horizontal room as possible, but don't allow any information to be truncated.
2. Display a table of processes that includes the process names and IDs. Also include columns for virtual and physical memory usage, expressing those values in megabytes (MB).
3. Use `Get-EventLog` to display a list of available event logs. (Hint: You'll need to read the help to learn the correct parameter to accomplish that.) Format the

output as a table that includes, in this order, the log display name and the retention period. The column headers must be "LogName" and "RetDays."

4. Display a list of services so that a separate table is displayed for services that are started and services that are stopped. Services that are started should be displayed first. (Hint: You'll use a `-groupBy` parameter).

## 8.11 Ideas for on your own

This is the perfect time to experiment with the formatting system. Try using the three main `Format-` cmdlets to create different forms of output. The labs in upcoming chapters will often ask you to use specific formatting, so you might as well hone your skills with these cmdlets and start memorizing the more-often-used parameters that we've covered in this chapter.

# 9 Filtering and comparisons

So far, we've been working with whatever output the shell gave us: all the processes, all the services, all of the event log entries, all of the hotfixes. That won't always be what you want, though. In many cases, you'll want to narrow the results down to a few items that specifically interest you. That's what you'll learn to do in this chapter.

## 9.1 Making the shell give you just what you need

The shell offers two broad models for narrowing down results, and they're both referred to as *filtering*. In the first model, you try to instruct the cmdlet that's retrieving information for you to only retrieve what you're specifically after. In the second model, you take everything that the cmdlet gives you and then use a second cmdlet to filter out the things you don't want.

Ideally, you'll use the first model, which I call *early filtering*, as much as possible. It may be as simple as telling the cmdlet what you're after. For example, with `Get-Service`, you can tell it which service names you want:

```
Get-Service -name e*,*s*
```

But if you want `Get-Service` to only return running services, regardless of their names, you can't tell the cmdlet to do that for you, because it doesn't offer any parameters to specify that.

Similarly, if you're using Microsoft's ActiveDirectory module, all of its `Get-` cmdlets support a `-filter` parameter. Although you can tell it `-filter *` to get all objects, doing so isn't recommended because of the load that can impose on a domain controller in large domains. Instead, you can specify criteria that explain precisely what you want:

```
Get-ADComputer -filter "Name -like '*DC'"
```

Once again, this technique is ideal because the cmdlet only has to retrieve matching objects. I also call this technique the *filter left* technique.

## 9.2 Filter left

*Filter left* simply means putting your filtering criteria as far to the left, or toward the beginning, of the command line as possible. The earlier you can filter out unwanted objects, the less work the remaining cmdlets on the command line will have to do, and the less unnecessary information that will have to be transmitted across the network to your computer.

The downside of the filter left technique is that every single cmdlet can implement its own means of specifying filtering, and every cmdlet will have varying abilities to do any filtering. With `Get-Service`, for example, you can pretty much only filter on the `Name` property of the services. With `Get-ADComputer`, however, you can filter on pretty much any Active Directory attribute that a `Computer` object might have. Being effective with the filter left technique requires you to learn a lot about how various cmdlets operate, which can mean a somewhat steeper learning curve. You'll benefit from better performance, though!

When you're not able to get a cmdlet to do all of the filtering you need, you'll turn to a core PowerShell cmdlet called `Where-Object` (which has an alias of `Where`). This uses a generic syntax and can be used to filter any kind of object, once you've retrieved it and put it into the pipeline.

To use `Where-Object`, you'll need to learn how to tell the shell what you want to filter, and that involves using the shell's comparison operators. Interestingly, some filter left techniques—such as the `-filter` parameter of the `Get-` cmdlets in the ActiveDirectory module—use the same comparison operators, so you'll be killing two birds with one stone. Some cmdlets, however (I'm thinking about `Get-WmiObject`, which we'll discuss later), use an entirely different filtering and comparison language, which we'll have to cover when we discuss those cmdlets.

## 9.3 Comparison operators

In computers, a *comparison* always involves taking two objects or values and testing their relationship to one another. You might be testing to see if they're equal, or to see if one is greater than another, or if one of them matches a text pattern of some kind. You indicate the kind of relationship you want to test by using a *comparison operator*. The result of the test is always a Boolean value: `True` or `False`. In other words, either the tested relationship is as you specified, or it isn't.

PowerShell uses the following comparison operators. Note that, when comparing text strings, these aren't case-sensitive. That means an uppercase letter is seen as equal to a lowercase letter.

- `-eq`—Equality, as in `5 -eq 5` (which is `True`) or `"hello" -eq "help"` (which is `False`)
- `-ne`—Not equal to, as in `10 -ne 5` (which is `True`) or `"help" -ne "help"` (which is `False`, because they are, in fact, equal, and we were testing to see if they were inequal)
- `-ge` and `-le`—Greater than or equal to, and less than or equal to, as in `10 -ge 5` (`True`) or `Get-Date -le '2012-12-02'` (which will depend on when you run this, and shows how dates can be compared in this fashion)
- `-gt` and `-lt`—Greater than and less than, as in `10 -lt 10` (`False`) or `100 -gt 10` (`True`)

For string comparisons, you can use a separate set of operators that are case-sensitive, if needed: `-ceq`, `-cne`, `-cgt`, `-clt`, `-cge`, `-cle`.

If you want to compare more than one thing at once, you can use the Boolean operators `-and` and `-or`. Each of those takes a subexpression on either side, and I usually enclose them in parentheses to make the line clearer to read:

- `(5 -gt 10) -and (10 -gt 100)` is `False`, because one or both subexpressions were `False`.
- `(5 -gt 10) -or (10 -lt 100)` is `True`, because at least one subexpression was `True`.

In addition, the Boolean `-not` operator simply reverses `True` and `False`. This can be useful when you're dealing with a variable or a property that already contains `True` or `False`, and you want to test for the opposite condition. For example, if I wanted to test whether a process was not responding, I could do this (I'm going to use `$_` as a placeholder for a process object):

```
$_.Responding -eq $False
```

Windows PowerShell defines `$False` and `$True` to represent the `False` and `True` Boolean values. Another way to write that comparison would be as follows:

```
-not $_.Responding
```

Because `Responding` normally contains `True` or `False`, the `-not` will reverse `False` to `True`. So if the process isn't responding (meaning `Responding` is `False`), my comparison will return `True`, indicating that the process is "not responding." I prefer the second technique because it reads, in English, more like what I'm actually testing for: "I want to see if the process is not responding." You'll sometimes see the `-not` operator abbreviated as an exclamation mark (`!`).

There are a couple of other comparison operators that are especially useful when you need to compare strings of text:

- `-like` accepts `*` as a wildcard, so you can compare to see if `"Hello" -like "*ll*"` (that would be `True`). `-notlike` is the reverse, and both are case-insensitive; use `-clike` and `-cnotlike` for case-sensitive comparisons.

- `-match` makes a comparison between a string of text and a regular expression pattern. `-notmatch` is its logical opposite, and as you might expect, `-cmatch` and `-cnotmatch` provide case-sensitive versions. Regular expressions are beyond the scope of what we'll cover in this book.

The neat thing about the shell is that you can test almost all of these right at the command line (the exception is the one where I used the `$_` placeholder—it won't work by itself, but you'll see where it will work in just a moment).

**TRY IT NOW** Go ahead and try any—or all—of these comparisons. Type them on a line and hit Return, like `5 -eq 5`, and see what you get.

### Above and beyond

If a cmdlet doesn't use the preceding PowerShell-style comparison operators, it probably uses the more traditional, programming language–style comparison operators that you might remember from high school or college (or even your daily work!):

- = equality
- <> inequality
- <= less than or equal to
- >= greater than or equal to
- > greater than
- < less than

If Boolean operators are supported, they're usually the words `AND` and `OR`; some cmdlets may support operators such as `LIKE` as well. You'll find support for all of these operators in the `-filter` parameter of `Get-WmiObject`, for example, and I'll repeat this list when we discuss that cmdlet in chapter 11.

Every cmdlet's designers get to pick how (and if) they'll handle filtering; you can often get examples of what they decided to do by reviewing the cmdlet's full help, including the usage examples near the end of the help file.

## 9.4 Filtering objects out of the pipeline

Once you've written a comparison, where do you use it? Well, using the comparison language I just outlined, you can use it with the `-filter` parameter of some cmdlets, perhaps most notably the ActiveDirectory module's `Get-` cmdlets. You can also use it with the shell's generic filtering cmdlet, `Where-Object`.

For example, want to get rid of all but the running services?

```
Get-Service | Where-Object -filter { $_.Status -eq 'Running' }
```

The `-filter` parameter is positional, so you'll often see this typed without it, and with the alias `Where`:

```
Get-Service | Where { $_.Status -eq 'Running' }
```

If you get used to reading that aloud, it sounds sensible: "where status equals running." Here's how it works: When you pipe objects to `Where-Object`, it examines each one of them using its filter. It places one object at a time into the `$_` placeholder and then runs the comparison to see if it's `True` or `False`. If it's `False`, the object is dropped from the pipeline. If the comparison is `True`, the object is piped out of `Where-Object` to the next cmdlet in the pipeline. In this case, the next cmdlet is `Out-Default`, which is always at the end of the pipeline (as we discussed in chapter 8) and which kicks off the formatting process to display your output.

That `$_` placeholder is a special creature: you've seen it used before (in chapters 7 and 8), and you'll see it in only one or two more contexts. You can only use this placeholder in the specific places where PowerShell looks for it, and this happens to be one of those places. As you learned in chapters 7 and 8, the period tells the shell that we're not comparing the entire object, but rather just one of its properties, `Status`.

Hopefully, you're starting to see where `Gm` comes in handy, as it gives you a quick and easy way to discover what properties an object has, so that you can turn around and use those properties in a comparison like this one. Always keep in mind that the column headers in PowerShell's final output don't always reflect the actual property names. For example, run `Get-Process` and you'll see a column like PM(MB); run `Get-Process | Gm` and you'll see that the actual property name is `PM`. That's an important distinction: always verify property names using `Gm`, not a `Format-` cmdlet.

## 9.5 The iterative command-line model

I want to go on a brief tangent with you and talk about what I call the PowerShell Iterative Command-Line Model, or PSICLM. There's no reason for it to have an acronym, but it's fun to try and pronounce it. The idea here is that you don't need to construct these large, complex command lines all at once and entirely from scratch. Start small.

Let's say I want to measure the amount of virtual memory being used by the ten most virtual memory–hungry processes. But if PowerShell itself is one of those processes, I don't want it included in the calculation. Let's take a quick inventory of what I need to do:

- Get processes
- Get rid of everything that's PowerShell
- Sort them by virtual memory
- Only keep the top 10 or bottom 10, depending on how I sorted them
- Add up the virtual memory for whatever is left

I believe you know how to do the first three of those. The fourth is accomplished with your old friend, `Select-Object`.

> **TRY IT NOW** Take a moment and read the help for `Select-Object`. Can you see any parameters that would enable you to keep just the first or last number of objects in a collection?

Hopefully you found the answer.

Finally, you need to add up the virtual memory. This is where you'll need to find a new cmdlet, probably by doing a wildcard search with `Get-Command` or `Help`. I might try the `Add` keyword, or the `Sum` keyword, or even the `Measure` keyword.

> **TRY IT NOW** See if you can find a command that would measure the total of a numeric property like virtual memory. Use `Help` or `Get-Command` with the `*` wildcard.

Hopefully you're trying these little tasks and not just reading ahead for the answer, because this is the key skill in making yourself a PowerShell expert! Once you think you have the answer, you might start in on the iterative approach.

To start with, I'll get processes. That's easy enough:

```
Get-Process
```

> **TRY IT NOW** Follow along in the shell, and run the same commands I'm running. After each, examine the output, and see if you can predict what I'll change for the next iteration of the command.

Next, I'll filter out what I don't want. Remember, "filter left" means I want to get the filter as close to the beginning of the command line as possible. In this case, I'm going to use `Where-Object` to do the filtering, so I want it to be the next cmdlet. That's not as good as having filtering occurring on the first cmdlet, but it's better than filtering later on down the pipeline.

In the shell, I'll hit the up arrow on the keyboard to recall my last command, and then add the next command:

```
Get-Process | Where-Object -filter { $_.Name -notlike 'powershell*' }
```

I'm not sure if it's "powershell" or "powershell.exe," so I used a wildcard comparison to cover all my bases. Any process that isn't like that name will remain in the pipeline.

I run that to test it, and then hit the up arrow again to add the next bit:

```
Get-Process | Where-Object -filter { $_.Name -notlike 'powershell*' } |
➥ Sort VM -descending
```

Hitting Return lets me check my work, and up arrow will let me add the next piece of the puzzle:

```
Get-Process | Where-Object -filter { $_.Name -notlike 'powershell*' } |
➥ Sort VM -descending | Select -first 10
```

Had I sorted in the default ascending order, I would have wanted to keep the `-last 10` before adding my last bit:

```
Get-Process | Where-Object -filter { $_.Name -notlike 'powershell*' } |
➥ Sort VM -descending | Select -first 10 |
➥ Measure-Object -property VM -sum
```

Hopefully you were able to figure out at least the name of that last cmdlet, if not the exact syntax I've used here.

This model—running a command, examining the results, recalling it, and modifying it for another try—is what differentiates PowerShell from more traditional scripting languages. As a command-line shell, you get those immediate results, and also the ability to quickly and easily modify your command if the results weren't what you wanted. Hopefully you're also seeing the power that you get by combining even the handful of cmdlets that you've learned so far.

## 9.6 Common points of confusion

Anytime I introduce `Where-Object` in a class, I usually come across two main sticking points. I tried to hit those pretty hard in the preceding discussion, but if there's any room left for doubt, let's clear it up now.

### 9.6.1 Filter left, please

You want your filtering criteria to go *as close to the beginning of the command line* as possible. If you can accomplish the filtering you need on the first cmdlet, do so; if not, try to filter in the second cmdlet so that the subsequent cmdlets have as little work to do as possible.

Also, try to accomplish filtering as close to the source of the data as possible. For example, if you're querying services from a remote computer and will need to use `Where-Object`—as I did in one of this chapter's examples—consider using PowerShell remoting to have the filtering occur on the remote computer, rather than bringing all of the object to your computer and filtering it there. You're going to tackle remoting in the next chapter, and I'll mention this idea of filtering at the source again there.

### 9.6.2 When $_ is allowed

The special `$_` placeholder is only valid in the places where PowerShell knows to look for it. When it's valid, it contains one object at a time from the ones that were piped into that cmdlet. Keep in mind that what's in the pipeline can and will change throughout the pipeline, as various cmdlets execute and produce output.

Also be careful of nested pipelines—the ones that occur inside a parenthetical command. For example, this can be tricky to figure out:

```
Get-Service -computername (Get-Content c:\names.txt |
➥ Where-Object -filter { $_ -notlike '*dc' }) |
➥ Where-Object -filter { $_.Status -eq 'Running' }
```

Let's walk through that:

- I started with `Get-Service`, but that isn't the first command that will execute. Because of the parentheses, `Get-Content` will execute first.
- `Get-Content` is piping its output—which consists of simple `String` objects—to `Where-Object`. That `Where-Object` is inside the parentheses, and within its filter, `$_` represents the `String` objects piped in from `Get-Content`. Only those strings that don't end in "dc" will be retained and output by `Where-Object`.
- The output of `Where-Object` becomes the result of the parenthetical command, because `Where-Object` was the last cmdlet inside the parentheses. So all

of the computer names that don't end in "dc" will be sent to the `-computername` parameter of `Get-Service`.

- *Now* `Get-Service` executes, and the `ServiceController` objects it produces will be piped to `Where-Object`. *That* instance of `Where-Object` will put one service at a time into its `$_` placeholder, and it will keep only those services whose `status` property is `Running`.

Sometimes I feel like my eyes are crossing with all the curly braces, periods, and parentheses—but that's how PowerShell works, and if you can train yourself to walk through the command carefully, you'll be able to figure out what it's doing.

## 9.7 Lab

Remember that `Where-Object` isn't the only way to filter, and it isn't even the one you should turn to first. I've kept this chapter a bit shorter so that you can have more time to work on hands-on examples, so following the principle of *filter left*, try to accomplish the following:

1. Import the ServerManager module in Windows Server 2008 R2. Using the `Get-WindowsFeature` cmdlet, display a list of server roles and features that are currently installed.
2. Import the ActiveDirectory module in Windows Server 2008 R2. Using the `Get-ADUser` cmdlet, display a list of users whose `PasswordLastSet` property is equal to the special value `$null`. (Hint: This property isn't retrieved from the directory by default. You'll have to specify a parameter that forces this property to be retrieved if you want to look at it). Your final list should include only the user name of the users who meet this criterion. This is a tricky task, because getting `$null` into the filter criteria for the cmdlet's own `-filter` parameter may not be possible.
3. Display a list of hotfixes that are security updates.
4. Using `Get-Service`, is it possible to display a list of services that have a start type of `Automatic`, but that aren't currently started?
5. Display a list of hotfixes that were installed by the Administrator, and which are updates.
6. Display a list of all processes running as either Conhost or Svchost.

## 9.8 Ideas for on your own

Practice makes perfect, so try filtering some of the output from the cmdlets you've already learned about, such as `Get-Hotfix`, `Get-EventLog`, `Get-Process`, `Get-Service`, and even `Get-Command`. For example, you might try and filter the output of `Get-Command` so that only cmdlets are shown. Or use `Test-Connection` to ping several computers, and only show the results from computers that did not respond. I'm not suggesting that you need to use `Where-Object` in every case, but you should practice using it when it's appropriate.

# 10 Remote control: one to one, and one to many

When I first started using PowerShell (in version 1), I was playing around with the `Get-Service` command, and noticed that it had a `-computerName` parameter. Hmmm … does that mean it can get services from other computers, too? After a bit of experimenting, I discovered that's exactly what it did. I got very excited and started looking for `-computerName` parameters on other cmdlets, and was disappointed to find that there were very few. A few more were added in v2, but the commands that have this parameter are vastly outnumbered by the commands that don't.

What I've realized since is that PowerShell's creators are a bit lazy—and that's a good thing! They didn't want to have to code a `-computerName` parameter for every single cmdlet, so they created a shell-wide system called *remoting*. Basically, it enables any cmdlet to be run on a remote computer. In fact, you can even run commands that exist on the remote computer but that don't exist on your own computer—meaning that you don't always have to install every single administrative cmdlet on your workstation. This remoting system is powerful, and it offers a number of interesting administrative capabilities.

## 10.1 The idea behind remote PowerShell

Remote PowerShell works somewhat similarly to Telnet and other age-old remote control technologies. When you run a command, it's actually running *on* the remote computer. Only the results of that command come back to your computer. Rather than using Telnet or SSH, however, PowerShell uses a new communications protocol called Web Services for Management (WS-MAN).

WS-MAN operates entirely over HTTP or HTTPS, making it easy to route through firewalls if necessary (because each of those protocols uses a single port to communicate). Microsoft's implementation of WS-MAN comes in the form of a background service, Windows Remote Management (WinRM). WinRM is installed along with PowerShell v2 and is started by default on server operating systems like Windows Server 2008 R2. It's installed on Windows 7 by default, but the service is disabled.

You've already learned that Windows PowerShell cmdlets all produce objects as their output. When you run a remote command, its output objects need to be put into a form that can be easily transmitted over a network using the HTTP (or HTTPS) protocol. XML, it turns out, is an excellent way to do that, so PowerShell automatically *serializes* those output objects into XML. The XML is transmitted across the network and is then *deserialized* on your computer back into objects that you can work with inside PowerShell.

Why should you care how this output is returned? Because those serialized objects are really just snapshots, of sorts; they don't update themselves continually. For example, if you were to get the objects that represent the processes running on a remote computer, what you'd get back would only be accurate for the exact point in time at which those objects were generated. Values like memory usage and CPU utilization won't be updated to reflect subsequent conditions. In addition, you can't tell the deserialized objects to do anything—you can't instruct one to stop itself, for example.

Those are basic limitations of remoting, but they don't stop you from doing some pretty amazing stuff. In fact, you can tell a remote process to stop itself—you just have to be a bit clever about it. I'll show you how in a bit.

There are two basic requirements to make remoting work:

- Both your computer and the one you want to send commands to must be running Windows PowerShell v2. Windows XP is the oldest version of Windows on which you can install PowerShell v2, so it's the oldest version that can participate in remoting.
- Ideally, both computers need to be members of the same domain, or of trusted/trusting domains. It's possible to get remoting to work outside of a domain, but it's tricky, and I won't be covering it in this chapter. To learn more about that scenario, open PowerShell and run `Help about_remote_troubleshooting`.

**TRY IT NOW** I'm hoping that you'll be able to follow along with some of the examples in this chapter. To do so, you'll ideally have a second test computer (or virtual machine) that's in the same Active Directory domain as the test computer you've been using up to this point. That second computer can be running any version of Windows, provided PowerShell v2 is installed. If you can't set up an additional computer or virtual machine, use "localhost" to create remoting connections to your current computer. You're still using remoting, but it isn't as exciting to be "remote controlling" the computer that you're sitting in front of.

## 10.2 WinRM overview

Let's talk a bit about WinRM, because you're going to have to configure it in order to start using remoting. Once again, you only need to configure WinRM—and PowerShell remoting—on those computers that will *receive* incoming commands. In most of the environments I've worked in, the administrators have enabled remoting on every Windows-based computer (keep in mind that PowerShell and remoting are supported all the way back to Windows XP). Doing so gives you the ability to remote into client desktop and laptop computers in the background (meaning the users of those computers won't know you're doing so), which can be tremendously useful.

WinRM isn't unique to PowerShell. In fact, it's likely that Microsoft will start using it for more and more administrative communications—even things that use other protocols today. With that in mind, Microsoft made WinRM able to route traffic to multiple administrative applications—not just PowerShell. WinRM essentially acts as a dispatcher: when traffic comes in, WinRM decides which application needs to deal with that traffic. All WinRM traffic is tagged with the name of a recipient application, and those applications must register with WinRM to listen for incoming traffic on their behalf. In other words, you'll not only need to enable WinRM, but you'll also need to tell PowerShell to register as an *endpoint* with WinRM.

One way to do that is to open a copy of PowerShell—making sure that you're running it as an Administrator—and run the `Enable-PSRemoting` cmdlet. You might sometimes see references to a different cmdlet, called `Set-WSManQuickConfig`. There's no need to run that one; `Enable-PSRemoting` will call it for you, and `Enable-PSRemoting` does a few extra steps that are necessary to get remoting up and running. All told, the cmdlet will start the WinRM service, configure it to start automatically, register PowerShell as an endpoint, and even set up a Windows Firewall exception to permit incoming WinRM traffic.

**TRY IT NOW** Go ahead and enable remoting on your second computer (or on the first one, if that's the only one you have to work with). Make sure you're running PowerShell as an Administrator (it should say "Administrator" in the window's title bar). If you're not, close the shell, right-click the PowerShell icon in the Start menu, and select Run as Administrator from the context menu.

If you're not excited about having to run around to every computer to enable remoting, don't worry: you can also do it with a Group Policy object (GPO), too. The necessary GPO settings are built into Windows Server 2008 R2 domain controllers (and you can download an ADM template from download.Microsoft.com to add these GPO settings to an older domain's domain controllers). Just open a Group Policy object and look under the Computer Configuration, then under Administrative Templates, then under Windows Components. Near the bottom of the list, you'll find both Remote Shell and Windows Remote Management. For now, I'm going to assume that you'll run `Enable-PSRemoting` on those computers that you want to configure, because at this point you're probably just playing around with a virtual machine or two.

> **NOTE** The about_remote_troubleshooting help topic in PowerShell provides more coverage on using GPOs. Look for the "How to enable remoting in an enterprise" and "How to enable listeners by using a Group Policy" sections within that help topic.

WinRM v2 (which is what PowerShell uses) defaults to using TCP port 5985 for HTTP and 5986 for HTTPS. Those ports help to ensure it won't conflict with any locally installed web servers, which tend to listen to 80 and 443 instead. You can configure WinRM to use alternative ports, but I don't recommend doing so. If you leave those ports alone, all of PowerShell's remoting commands will run normally. If you change the ports, you'll have to always specify an alternative port when you run a remoting command, which just means more typing for you.

If you absolutely must change the port, you can do so by running this command:

```
Winrm set winrm/config/listener?Address=*+Transport=HTTP
➥ @{Port="1234"}
```

In this example, "1234" is the port you want. Modify the command to use HTTPS instead of HTTP to set the new HTTPS port.

> **DON'T TRY IT NOW** Although you may want to change the port in your production environment, don't change it on your test computer. Leave WinRM using the default configuration so that the remainder of this book's examples will work for you without modification.

I should admit that there is a way to configure WinRM on client computers to use alternative default ports, so that you're not constantly having to specify an alternative port when you run commands. But for now let's stick with the defaults Microsoft came up with.

> **NOTE** If you do happen to browse around in the Group Policy object settings for Remote Shell, you'll notice that you can set things like how long a remoting session can sit idle before the server kills it, how many concurrent users can remote into a server at once, how much memory and how many processes each remote shell can utilize, and the maximum number of remote shells a given user can open at once. These are all great ways to help ensure that your servers don't get overly burdened by forgetful administrators! By default, however, you *do* have to be an Administrator to use remoting, so you don't need to worry about ordinary users clogging up your servers.

## 10.3 Using Enter-PSSession and Exit-PSSession for 1:1 remoting

PowerShell uses remoting in two distinct ways. The first is called *one-to-one,* or 1:1, remoting (the second way is one-to-many remoting, and you'll see it in the next section). With this kind of remoting, you're basically accessing a shell prompt on a single remote computer. Any commands you run will run directly on that computer, and you'll see results in the shell window. This is vaguely similar to using Remote Desktop Connection, except that you're limited to the command-line environment of Windows PowerShell. Oh, and this kind of remoting uses a *fraction* of the resources that Remote Desktop requires, so it imposes much less overhead on your servers!

To establish a one-to-one connection with a remote computer, run this command:

```
Enter-PSSession -computerName Server-R2
```

Of course, you'll need to provide the correct computer name instead of `Server-R2`.

Assuming you enabled remoting on that computer, that you're all in the same domain, and that your network is functioning correctly, you should get a connection going. PowerShell lets you know that you've succeeded by changing the shell prompt:

```
[server-r2] PS C:\>
```

That prompt tells you that everything you're doing is taking place on Server-R2 (or whatever server you connected to). You can run whatever commands you like. You can even import any modules, or add any PSSnapins, that happen to reside on that remote computer.

> **TRY IT NOW** Go ahead and try to create a remoting connection to your second computer or virtual machine. If you haven't yet done so, you'll need to enable remoting on that computer before you try to connect to it. Note that you're going to need to know the real computer name of the remote computer; WinRM won't, by default, permit you to connect by using its IP address or a DNS alias.

Even your permissions and privileges carry over across the remote connection. Your copy of PowerShell will pass along whatever security token it's running under (it does this with Kerberos, so it doesn't pass your username or password across the network). Any command you run on the remote computer will run under your credentials, so you'll be able to do anything you'd normally have permission to do. It's just like logging directly into that computer's console and using its copy of PowerShell directly.

Well, almost. There are a couple of differences:

- Even if you have a PowerShell profile script on the remote computer, it won't run when you connect using remoting. We haven't fully covered profile scripts yet (they're in chapter 24), but suffice to say that they're a batch of commands that run automatically each time you open the shell. Folks use them to automatically load shell extensions and modules and so forth. That doesn't happen when you remote into a computer, so be aware of that.

- You're still restricted by the remote computer's execution policy. Let's say your local computer's policy is set to `RemoteSigned`, so that you can run local, unsigned scripts. That's great, but if the remote computer's policy is set to the default, `Restricted`, it won't be running any scripts for you when you're remoting into it.

Aside from those two fairly minor caveats, you should be good to go. Oh, wait—what do you do when you're done running commands on the remote computer? Many PowerShell cmdlets come in pairs, with one cmdlet doing something and the other doing the opposite. In this case, if `Enter-PSSession` gets you *into* the remote computer, can you guess what would get you *out* of the remote computer? If you guessed `Exit-PSSession`, give yourself a prize. The command doesn't need any parameters; just run it and your shell prompt will change back to normal, and the remote connection will close automatically.

> **TRY IT NOW** Go ahead and exit the remoting session, if you created one. We're done with it for now.

What if you forget to run `Exit-PSSession` and instead close the PowerShell window? Don't worry. PowerShell and WinRM are smart enough to figure out what you did, and the remote connection will close all by itself.

I do have one caution to offer. When you're remoting into a computer, don't run `Enter-PSSession` *from that computer* unless you fully understand what you're doing. Let's say you work on Computer A, which runs Windows 7. You remote into Server-R2. Then, at the PowerShell prompt, you run this:

```
[server-r2] PS C:\>enter-pssession server-dc4
```

Now, Server-R2 is maintaining an open connection to Server-DC4. That can start to create a "remoting chain" that's hard to keep track of, and which imposes unnecessary overhead on your servers. There are times when you might *have* to do this—I'm thinking mainly of instances where a computer like Server-DC4 sits behind a firewall and you can't access it directly, so you use Server-R2 as a middleman to hop over to Server-DC4. But, as a general rule, try to avoid remote chaining.

When you're using this one-to-one remoting, you don't need to worry about objects being serialized and deserialized. As far as you're concerned, you're typing directly on the remote computer's console. If you retrieve a process and pipe it to `Stop-Process`, it'll stop as you would expect it to.

## 10.4 Using Invoke-Command for one-to-many remoting

The next trick—and honestly, this is one of the coolest things in Windows PowerShell—is to send a command to *multiple remote computers at the same time*. That's right, full-scale distributed computing. Each computer will independently execute the command and send the results right back to you. It's all done with the `Invoke-Command` cmdlet, and it's called *one-to-many*, or 1:n, remoting.

The command looks something like this:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
➥ -command { Get-EventLog Security -newest 200 |
➥ Where { $_.EventID -eq 1212 }}
```

**TRY IT NOW** Go ahead and run this command. Substitute the name of your remote computer (or computers) where I've put my three computer names.

Everything in those outermost curly braces, the `{}`, will get transmitted to the remote computers—all three of them. By default, PowerShell will talk to up to 32 computers at once; if you specified more than that, it will queue them up, so that as one computer completes, the next one in line will begin. If you have an awesome network and powerful computers, you could raise that number by specifying the `-throttleLimit` parameter of `Invoke-Command`—read the command's help for more information.

### Be careful about the punctuation

We need to pause for a moment and dig into the preceding example command, because this is a case where PowerShell's punctuation can get confusing, and that confusion can make you do the wrong thing when you start constructing these command lines on your own.

There are two commands in that example that use curly braces: `Invoke-Command` and `Where` (which is an alias for `Where-Object`). `Where` is entirely nested within the outer set of braces. The outermost set of braces enclose everything that's being sent to the remote computers for execution:

```
Get-EventLog Security -newest 200 | Where { $_.EventID -eq 1212 }
```

It can be tough to follow that nesting of commands, especially in a book like this where the physical width of the page makes it necessary to display the command across several lines of text.

Don't read any further until you're sure you can identify the exact command that's being sent to the remote computer, and that you understand what each matched set of curly braces is for.

I should tell you that you won't see the `-command` parameter in the help for `Invoke-Command`—but the command I just showed you will work fine. The `-command` parameter is an *alias,* or nickname, for the `-scriptblock` parameter that you *will* see listed in the help. I have an easier time remembering `-command`, so I tend to use it instead of `-scriptblock`, but they both work the same way.

If you read the help for `Invoke-Command` carefully (see how I'm continuing to push those help files?), you'll also notice a parameter that lets you specify a script file, rather than a command. That parameter lets you send an entire script from your local computer to the remote computers—meaning you can automate some pretty complex tasks and have each computer do its own share of the work.

> **TRY IT NOW** Make sure you can identify the `-scriptblock` parameter in the help for `Invoke-Command`, and that you can spot the parameter that would enable you to specify a file path and name instead of a script block.

I want to circle back to the `-computerName` parameter for a bit. When I first used `Invoke-Command`, I typed a comma-separated list of computer names, just as I did in the previous example. But I work with a *lot* of computers, so I didn't want to have to type them all in every time. I keep text files for some of my common computer categories, like web servers and domain controllers. Each text file contains one computer name per line, and that's it—no commas, no quotes, no nothing. PowerShell makes it easy for me to use those files:

```
Invoke-Command -command { dir }
➥ -computerName (Get-Content webservers.txt)
```

The parentheses here force PowerShell to execute `Get-Content` first—pretty much the same way parentheses work in math. The results of `Get-Command` are then stuck into the `-computerName` parameter, which then works against each of the computers that are listed in the file.

I also sometimes want to query computer names from Active Directory. This is a bit trickier. I can use the `Get-ADComputer` command (from the ActiveDirectory module in Windows Server 2008 R2) to retrieve computers, but I can't stick that command in parentheses like I did with `Get-Content`. Why not? Because `Get-Content` produces simple strings of text, which `-computerName` is expecting. `Get-ADComputer`, on the other hand, produces entire computer objects, and the `-computerName` parameter won't know what to do with them.

If I want to use `Get-ADComputer`, I need to find a way to get just the *values* from those computer objects' `Name` properties. Here's how:

```
Invoke-Command -command { dir } -computerName (
➥ Get-ADComputer -filter * -searchBase "ou=Sales,dc=company,dc=pri" |
➥ Select-Object -expand Name )
```

> **TRY IT NOW** If you're running PowerShell on a Windows Server 2008 R2 domain controller, or on a Windows 7 computer that has the Remote Server Administration Tools installed, you can run `Import-Module ActiveDirectory` and then try the preceding command. If your test domain doesn't have a Sales OU that contains a computer account, then change `ou=Sales` to `ou=Domain Controllers`, and be sure to change `company` and `pri` to the appropriate values for your domain (for example, if your domain is mycompany.org, you would substitute `mycompany` for `company` and `org` for `pri`).

Within the parentheses, I've piped the computer objects to `Select-Object`, and I've used its `-expand` parameter. I'm telling it to expand the `Name` property of whatever came in—in this case, those computer objects. The result of that entire parenthetical expression will be a bunch of computer names, not computer objects—and computer names are exactly what the `-computerName` parameter wants to see.

Just to be complete, I should mention that the `-filter` parameter of `Get-ADComputer` specifies that all computers should be included in the command's output. The `-searchBase` parameter tells the command to start looking for computers in the specified location—in this case, the Sales OU of the company.pri domain. The `Get-ADComputer` command is only available on Windows Server 2008 R2, and on Windows 7 after installing the Remote Server Administration Tools (RSAT). On those operating systems, you have to run `Import-Module ActiveDirectory` to load the Active Directory cmdlets into the shell so that they can be used.

## 10.5 Differences between remote and local commands

I want to explain a bit about the differences between running commands using `Invoke-Command`, and running those same commands locally, as well as the differences between remoting and other forms of remote connectivity. For this entire discussion, I'll use this command as my example:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
➥ -command { Get-EventLog Security -newest 200 |
➥ Where { $_.EventID -eq 1212 }}
```

Let's look at some alternatives, and why they're different.

### 10.5.1 Invoke-Command versus -ComputerName

Here's an alternative way to perform that same basic task:

```
Get-EventLog Security -newest 200
➥ -computerName Server-R2,Server-DC4,Server12
➥ | Where { $_.EventID -eq 1212 }
```

Here, I've used the `-computerName` parameter of `Get-EventLog`, rather than invoking the entire command remotely. I'll get more or less the same results, but there are some important differences in how the command executes:

- Using this command, the computers will be contacted sequentially rather than in parallel, which means the command may take longer to execute.
- The output won't include a `PSComputerName` property, which may make it harder for me to tell which result came from which computer.
- The connection won't be made using WinRM, but will instead use whatever underlying protocol the .NET Framework decides on. I don't know what that is, and it might be harder to get the connection through any firewalls that are between me and the remote computer.
- I'm querying 200 records from each of the three computers, and only then am I filtering through them to find the ones with `EventID 1212`. That means I am probably bringing over a lot of records that I don't want.

- I'm getting back event log objects that are fully functional.

These differences apply to any cmdlet that has a `-computerName` parameter. Generally speaking, it can be more efficient and effective to use `Invoke-Command` rather than a cmdlet's `-computerName` parameter.

Here's what would have happened if I'd used the original `Invoke-Command` instead:

- The computers would have been contacted in parallel, meaning the command could complete somewhat more quickly.
- The output would have included a `PSComputerName` property, enabling me to more easily distinguish the output from each computer.
- The connection would have been made through WinRM, which uses a single, predefined port that can be easier to get through any intervening firewalls.
- Each computer would have queried the 200 records and filtered them *locally*. The only data transmitted across the network would have been the result of that filtering, meaning that only the records I cared about would have been transmitted.
- Before transmitting, each computer would have serialized its output into XML. My computer would have received that XML and deserialized it back into something that looks like objects. But they wouldn't have been real event log objects, and that might limit what I could do with them once they were on my computer.

That last point is a big distinction between using a `-computerName` parameter and using `Invoke-Command`. Let's discuss that distinction.

### 10.5.2 Local versus remote processing

Here's my original example again:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
➥ -command { Get-EventLog Security -newest 200 |
➥ Where { $_.EventID -eq 1212 }}
```

Now, compare it to this alternative:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
➥ -command { Get-EventLog Security -newest 200 } |
➥ Where { $_.EventID -eq 1212 }
```

The differences are subtle. Actually, there's only one difference: I moved one of those curly braces.

In the second version, only `Get-EventLog` is being invoked remotely. All of the results generated by `Get-EventLog` will be serialized and sent to my computer, where they'll be deserialized into objects and then piped to `Where` and filtered. The second version of the command is less efficient, because a lot of unnecessary data is being transmitted across the network, and my one computer is having to filter the results from three computers, rather than those three computers filtering their own results for me. The second version, in other words, is a bad idea.

Let's look at two versions of another command. Here's the first:

```
Invoke-Command -computerName Server-R2
➥ -command { Get-Process -name Notepad } |
➥ Stop-Process
```

Here's the second version:

```
Invoke-Command -computerName Server-R2
➥ -command { Get-Process -name Notepad |
➥ Stop-Process }
```

Once again, the only difference between these two is the placement of a curly brace. In this example, however, the first version of the command won't work.

Look carefully: I'm sending `Get-Process -name Notepad` to the remote computer. The remote computer retrieves the specified process, serializes it into XML, and sends it to me across the network. My computer receives that XML, deserializes it back into an object, and pipes it to `Stop-Process`. The problem is that the deserialized XML doesn't contain enough information for my computer to realize that the process *came from a remote machine*. Instead, my computer will try to stop the Notepad process *running locally*, which isn't what I wanted at all.

The moral of the story is to always complete as much of your processing on the remote computer as possible. The only thing you should expect to do with the results of `Invoke-Command` is to display them or store them, as a report or data file or something. The second version of my command follows that advice: what's being sent to the remote computer is `Get-Process -name Notepad | Stop-Process`, so the entire command—both getting the process and stopping it—happens on the remote computer. Because `Stop-Process` doesn't normally produce any output, there won't be any objects to serialize and send to me, so I won't see anything on my local console. But the command will do what I want: stop the Notepad process *on the remote computer*, not on my local machine.

Whenever I use `Invoke-Command`, I always look at the commands after it. If I see commands for formatting, or for exporting data, I'm fine, because it's okay to do those things with the results of `Invoke-Command`. But if `Invoke-Command` is followed by action cmdlets—ones that start, stop, set, change, or do something else—then I sit back and try to think about what I'm doing. Ideally, I want all of those actions to happen on the remote computer, not on my local computer.

## 10.6 But wait, there's more

These examples have all used ad hoc remoting connections, meaning that I specified computer names. If you're going to be reconnecting to the same computer (or computers) several times within a short period of time, you can create reusable, persistent connections to use instead. We'll cover that technique in chapter 18.

I should also acknowledge that not every company is going to allow PowerShell remoting to be enabled—at least, not right away. Companies with extremely restrictive security policies may, for example, have firewalls on all client and server computers, which would block the remoting connection. If your company is one of those, see if an exception is in place for Remote Desktop Protocol (RDP). I find that's a common exception, because Administrators obviously need some remote connectivity to servers. If RDP is allowed, try to make a case for PowerShell remoting. Remoting connections can be audited (they look like network logins, much like accessing a file share would appear in the audit log), and they're locked down by default to only permit Administrators to connect. It's not that different from RDP in terms of security risks, and it imposes much less overhead on the remote machines than RDP does.

## 10.7 Common points of confusion

Whenever we start using remoting in a class that I'm teaching, there are some common problems that crop up over the course of the day:

- Remoting only works, by default, with the remote computer's real computer name. You can't use DNS aliases or IP addresses.
- Remoting is designed to be more or less automatically configuring within a domain. If every computer involved, and your user account, all belong to the same domain (or trusting domains), things will work great. If not, you'll need to run `help about_remote_troubleshooting` and dig into the details.
- When you invoke a command, you're asking the remote computer to launch PowerShell, run your command, and then close PowerShell. The next command you invoke on that same remote computer will be starting from scratch—anything that was run in the first invocation will no longer be in effect. If you need to run a whole series of related commands, put them all into the same invocation.
- Make absolutely certain that you're running PowerShell as an Administrator, especially if your computer has User Account Control (UAC) enabled. If the account you're using doesn't have Administrator permissions on the remote computer, then use the `-credential` parameter of `Enter-PSSession` or `Invoke-Command` to specify an alternative account that does have Administrator permissions.
- If you're using a local firewall product other than the Windows Firewall, `Enable-PSRemoting` won't set up the necessary firewall exceptions. You'll need to do so manually. If your remoting connection will need to traverse a regular firewall, such as one implemented on a router or proxy, then it'll also need a manually entered exception for the remoting traffic.
- Don't forget that any settings in a Group Policy object (GPO) override anything you configure locally. I've seen administrators struggle for hours to get remoting working, only to finally discover that a GPO was overriding everything they did. In some cases, that GPO was put into place a long time ago by a well-meaning colleague, who had long since forgotten it was there. Don't assume that there's no GPO affecting you; check and see for sure.

## 10.8 Lab

It's time to start combining some of what you've learned about remoting with what you've learned in previous chapters. See if you can accomplish these tasks:

1. Make a one-to-one connection with a remote computer. Launch Notepad.exe. What happens?
2. Using `Invoke-Command`, retrieve a list of services that aren't started from one or two remote computers. Format the results as a wide list. (Hint: It's okay to retrieve results and have the formatting occur on your computer—don't include the `Format-` cmdlet in the commands that are invoked remotely).
3. Use `Invoke-Command` to get a list of the top ten processes for virtual memory (VM) usage. Target one or two remote computers, if you can.
4. Create a text file that contains three computer names, with one name per line. It's okay to use the same computer name three times if you only have access to one remote computer. Then use `Invoke-Command` to retrieve the 100 newest Application event log entries from the computer names listed in that file.

## 10.9 Ideas for on your own

One of the PowerShell modules included in Windows 7 is TroubleshootingPack, which provides command-line access to the new troubleshooting pack functionality in the operating system. I always tell my students and clients to consider enabling PowerShell remoting on all of their client computers, in part because it gives you remote command-line access to those troubleshooting packs. When a user calls for help, rather than walking them through a wizard over the phone, you can just remote in and run the same wizard, in command-line form rather than GUI form, yourself.

If you have access to a remote Windows 7 computer, enable remoting on it. Initiate a one-to-one session and import the TroubleshootingPack module. Then see if you can get and invoke a troubleshooting pack. Remember, run `get-command -module troubleshootingpack` to see a list of cmdlets in that module (there are only two), and run help on those cmdlets to see how they work. You have to provide a file path to the troubleshooting pack you want; you'll find them in \Windows\Diagnostics by default.

Being able to remotely execute these troubleshooting packs—which can even take corrective action if a problem is found—is a strong argument for enabling remoting on client computers, especially those running Windows 7.

# 11 Tackling Windows Management Instrumentation

I've been looking forward to writing this chapter, and dreading it at the same time. Windows Management Instrumentation (WMI) is probably one of the best things Microsoft has ever offered to administrators. At the same time, it's also one of the worst things they've ever inflicted on us. In this chapter, I'll be introducing you to WMI, showing you how it works, and explaining some of its less-beautiful aspects, so that there's full disclosure on what you're up against.

## 11.1 Retrieving management information

The idea behind WMI is a good one: it's a generic system for retrieving management information. In some limited cases it can also be used for implementing configuration changes, although, for the most part, Microsoft hasn't leveraged that well or consistently.

WMI is built primarily around a system of *providers*, and each provider is designed to expose a particular type of management information. For example, on Windows Server, when you install the DNS Server role, you also install the bits that make DNS accessible through WMI, enabling you to query DNS records. Windows has a number of providers that install by default and provide information about the core operating system and computer hardware. Each computer can have a completely different set of WMI providers, because each computer on your network will have different software installed.

Like everything in PowerShell, WMI presents its information in the form of objects, and those objects have properties (and sometimes methods). The properties contain the management information you might be interested in; any methods available can be used to initiate actions or to make configuration changes.

## 11.2 A WMI primer

A typical Windows computer will contain tens of thousands of pieces of management information, and WMI seeks to organize that into something that's approachable and more or less sensible.

At the top level, WMI is organized into *namespaces*. A namespace is really just a sort of folder that ties to a specific product or technology. For example, the root\CIMv2 namespace contains all the Windows operating system and computer hardware information; the root\MicrosoftDNS namespace includes all the information about DNS Server (assuming you've installed that role on the computer). On client computers, root\SecurityCenter contains information about firewall, antivirus, and antispyware utilities.

Within a namespace, WMI is divided into a series of *classes*. A class represents a management component that WMI knows how to query. For example, the AntivirusProduct class in root\SecurityCenter is designed to hold information about antispyware products; the Win32_LogicalDisk class in root\CIMv2 is designed to hold information about logical disks. But just because a class exists on a computer doesn't mean that the computer actually has any of those components: the Win32_TapeDrive class is present on all versions of Windows, whether or not a tape drive is actually installed.

When you do have one or more manageable components, you'll have an equal number of *instances* for that class. An instance is simply a real-world occurrence of something represented by a class. If your computer has a single BIOS (and they all do), you'll have one instance of Win32_BIOS in root\CIMv2; if your computer has a hundred background services installed, you'll have a hundred instances of Win32_Service. Note that the class names in root\CIMv2 tend to start with either Win32_ (even on 64-bit machines) or CIM_ (which stands for Common Information Model, the standard upon which WMI is built). In other namespaces, those class name prefixes aren't usually used. Also, it's possible for class names to be duplicated across namespaces. It's rare, but WMI allows for it, because each namespace acts as a kind of container and boundary. When you're referring to a class, you'll also have to refer to its namespace, so that WMI knows where to look for the class and so that it doesn't get confused between two classes that have the same name but live in different namespaces.

On the surface, using WMI seems fairly simple: you figure out which class contains the information you want, query that class's instances from WMI, and then examine the instances' properties to see the management information. In some cases, you may ask an instance to execute a method in order to initiate an action or start a configuration change.

## 11.3 The bad news about WMI

Unfortunately, for most of its life (the situation has recently changed), Microsoft didn't exercise a lot of internal controls over WMI. They established a set of programming standards, but the product groups were more or less left to their own devices for how they implemented classes and whether or not they chose to document them. The result is that WMI can be a confusing mishmash.

Within the root\CIMv2 namespace, for example, very few classes have any methods that allow you to change configuration settings. Properties are read-only, meaning that you must have a method to make changes; if a method doesn't exist, you can't use WMI to make changes for that class. When the IIS team adopted WMI (for IIS version 6), they implemented parallel classes for a lot of elements. A website, for example, could be represented by one class that had the typical read-only properties, but also by a second class that had writable properties that you could change. Very confusing—and the confusion was made worse by the fact that there wasn't any good documentation on how to use those classes, because the IIS team originally intended them to be used mainly by their own tools, not directly by administrators.

There's no rule saying that a product has to use WMI, or that if it does use WMI that it must expose every possible component through WMI. Microsoft's DHCP server is inaccessible to WMI, as is its old WINS server. Although you can query the configuration of a network adapter, you can't retrieve its link speed, because that information isn't supplied. Although most of the Win32_ classes are well documented, few of the classes in other namespaces are documented at all. WMI isn't searchable, so the process of finding the class that you need can be time-consuming and frustrating (although I'll try to help with that in the next section).

The WMI repository—the place where Windows keeps all the WMI information—can also become corrupted, and that seems to occur a lot more on client computers than on servers. You might not even notice the problem unless you're using System Center Configuration Manager, which relies heavily on WMI and can't inventory computers properly when the repository becomes corrupted. If you find yourself in possession of a corrupted repository, check out the "Repairing and re-registering the WMI" article on Ramesh Srinivasan's *Troubleshooting Windows* blog (http://windowsxp.mvps.org/repairwmi.htm), which provides a good overview of what steps to take, and in what order. Rebuilding the repository isn't ever a good first step, but it's sometimes necessary, and that article will walk you through the process. You can also hit your favorite search engine with a search like "wmi repository corrupted" and you'll get a number of useful links and tools to try. Microsoft claims to have corrected the major corruption issues in Windows 7.

The good news is that Microsoft is making an effort to provide PowerShell cmdlets for as many administration tasks as possible. For example, WMI used to be the only practical way to programmatically restart a remote computer, using a method of the `Win32_OperatingSystem` class. Now, PowerShell provides a `Restart-Computer` cmdlet.

In some cases, cmdlets will use WMI internally, but you won't have to deal with WMI directly in those cases. Cmdlets can provide a more consistent interface for you, and they're almost always better documented. WMI isn't going away, but over time you'll probably have to deal with it—and its eccentricities—a lot less.

## 11.4 Exploring WMI

Perhaps the easiest way to get started with WMI is to put PowerShell aside for a second and explore WMI on its own. I use a free WMI Explorer tool that I downloaded from Sapien (http://www.primaltools.com/downloads/communitytools/); the tool doesn't require installation, which means you can easily copy it to a USB flash drive and carry it to whatever computer you're interested in. Because each computer can have different WMI stuff, you'll want to run the tool directly on whatever computer you're planning to query, so that you can see that computer's WMI repository.

I locate most of what I need in WMI with this tool. It does require a lot of browsing and patience—I'm not pretending this is a perfect process—but it eventually gets me there. Figure 11.1 shows an example.

Let's say I needed to query a bunch of client computers and see what their icon spacing is set to. That's something that has to do with the Windows desktop, and

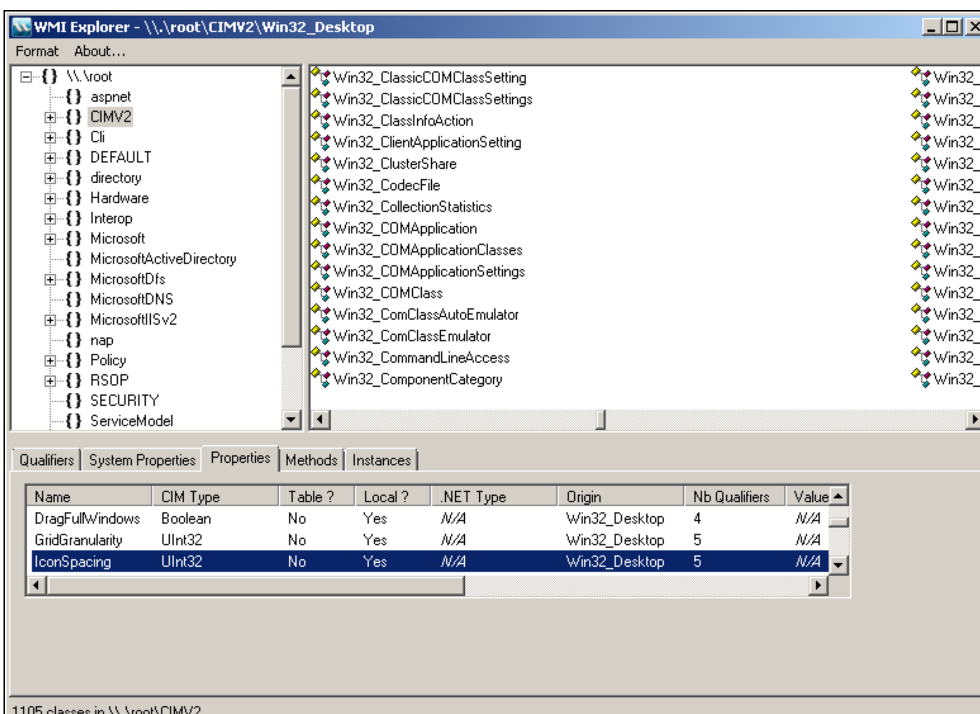



**Figure 11.1 Using the WMI Explorer to locate a WMI class**

that's a core part of the operating system, so I started in the root\CIMV2 class, shown in the tree view on the left side of the WMI Explorer. Clicking the namespace brings up a list of its classes in the right side, and I took a guess on "Desktop" as a keyword. Scrolling to the right, I eventually found Win32_Desktop and clicked on that. Doing so enables the details pane at the bottom, and I clicked on the Properties tab to see what was available. About a third of the way down, I found IconSpacing, which is listed as an integer.

Here's the trick that most people forget: once you've found a class and the property or properties you want, click on the Instances tab. There, as shown in figure 11.2, I can see that there are several instances for this class. It looks as if there's one instance for each user account on the computer, in fact. That makes sense, because each user will have their own desktop configuration, and each might select a different icon spacing setting. So when I query this, I'll either need to specify the exact instance I want, or I'll need to get all of the instances and then decide which one's icon spacing matters to me.

Obviously, search engines are another good way to find the class you want. I tend to prefix queries with "wmi," as in "wmi icon spacing," and that will often pull up an

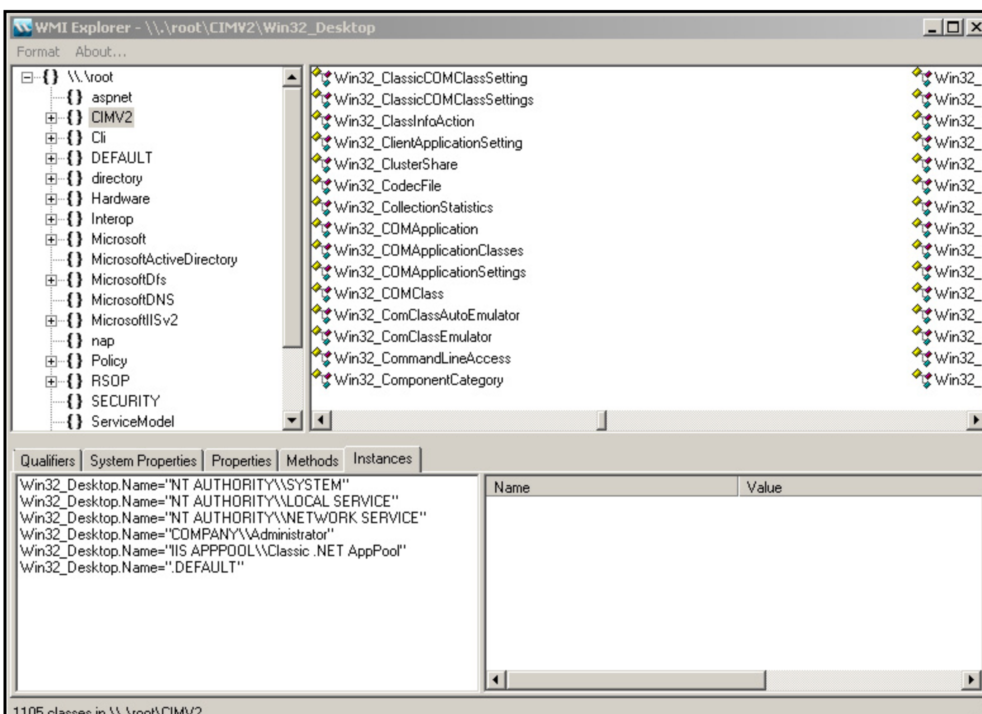



**Figure 11.2 Reviewing the available instances for the `Win32_Desktop` class**

example or two that points me in the right direction. The example might be VBScript-related, or might even be in a .NET language like C# or Visual Basic, but that's okay because I'm only after the WMI class name. For example, I just searched for "wmi icon spacing" and turned up http://stackoverflow.com/questions/202971/formula-or-api-for-calculating-desktop-icon-spacing-on-windows-xp as the first result. On that page I found some C# code:

```
ManagementObjectSearcher searcher = new ManagementObjectSearcher
     ("root\\CIMV2","SELECT * FROM Win32_Desktop");
```

I've no idea what any of that means, but `Win32_Desktop` looks like a WMI class name. My next search will be for that class name, as such a search will often turn up whatever documentation may exist. I'll cover the documentation a bit later in this chapter.

## 11.5 Using Get-WmiObject

PowerShell only makes you learn a single cmdlet to retrieve anything you want from WMI: `Get-WmiObject`. With it, you can specify a namespace, a class name, and even the name of a remote computer—and alternative credentials, if needed—to retrieve all instances of that class from the computer specified.

You can even provide filter criteria if you want fewer than all of the instances of the class. You can get a list of classes from a namespace. Here's the syntax for that:

```
Get-WmiObject -namespace root\cimv2 -list
```

Note that namespace names use a backslash, not a forward slash. To retrieve a class, specify the namespace and class name:

```
Get-WmiObject -namespace root\cimv2 -class win32_desktop
```

The root\CIMv2 namespace is the system default namespace on Windows XP Service Pack 2 and later, so if your class is in that namespace, you don't need to specify it. Also, the `-class` parameter is positional, so if you provide the class name in the first position, the cmdlet will work exactly the same.

Here are two examples, including one that uses the `Gwmi` alias instead of the full cmdlet name:

```
PS C:\> Get-WmiObject win32_desktop
PS C:\> gwmi antivirusproduct -namespace root\securitycenter
```

> **TRY IT NOW** You should start following along at this point, running each of the commands I show you. For commands that include a remote computer name, you can substitute `localhost` if you don't have another remote computer that you can test against.

For many WMI classes, PowerShell has configuration defaults that specify which properties are shown. `Win32_OperatingSystem` is a good example because it only displays six of its properties, in a list, by default. Keep in mind that you can always pipe the

WMI objects to `Gm` or to `Format-List *` to see all of the available properties; `Gm` will also list available methods. Here's an example:

```
PS C:\> gwmi win32_operatingsystem | gm

   TypeName: System.Management.ManagementObject#root\cimv2\Win32_Operating
System

Name                              MemberType        Definition
----                              ----------        ----------
Reboot                            Method            System.Managemen...
SetDateTime                       Method            System.Managemen...
Shutdown                          Method            System.Managemen...
Win32Shutdown                     Method            System.Managemen...
Win32ShutdownTracker              Method            System.Managemen...
BootDevice                        Property          System.String Bo...
BuildNumber                       Property          System.String Bu...
BuildType                         Property          System.String Bu...
Caption                           Property          System.String Ca...
CodeSet                           Property          System.String Co...
CountryCode                       Property          System.String Co...
CreationClassName                 Property          System.String Cr...
```

I've truncated that output to save space, but you'll see the whole thing if you run the same command.

The `-filter` parameter lets you specify criteria for retrieving specific instances. This can be a bit tricky to use, so here's an example of the worst-case usage:

```
PS C:\> gwmi -class win32_desktop -filter "name='COMPANY\\Administrator'"

__GENUS                : 2
__CLASS                : Win32_Desktop
__SUPERCLASS           : CIM_Setting
__DYNASTY              : CIM_Setting
__RELPATH              : Win32_Desktop.Name="COMPANY\\Administrator"
__PROPERTY_COUNT       : 21
__DERIVATION           : {CIM_Setting}
__SERVER               : SERVER-R2
__NAMESPACE            : root\cimv2
__PATH                 : \\SERVER-R2\root\cimv2:Win32_Desktop.Name="COMPANY
                         \\Administrator"
BorderWidth            : 1
Caption                :
CoolSwitch             :
CursorBlinkRate        : 530
Description            :
DragFullWindows        : False
GridGranularity        :
IconSpacing            : 43
IconTitleFaceName      : Tahoma
IconTitleSize          : 8
IconTitleWrap          : True
Name                   : COMPANY\Administrator
Pattern                : 0
ScreenSaverActive      : False
```

```
ScreenSaverExecutable :
ScreenSaverSecure     :
ScreenSaverTimeout    :
SettingID             :
Wallpaper             :
WallpaperStretched    : True
WallpaperTiled        : False
```

There are some things you should notice about this command and its output:

- The filter criteria is usually enclosed in double quotation marks.
- The filter comparison operators aren't the normal PowerShell `-eq` or `-like` operators. Instead, WMI uses more traditional, programming-like operators, such as `=`, `>`, `<`, `<=`, `>=`, and `<>`. You can use the keyword `LIKE` as an operator, and when you do your comparison value can use `%` as a character wildcard, as in `"NAME LIKE '%administrator%'"`.
- Any string comparison values are enclosed in single quotation marks, which is why the outermost quotes that contain the entire filter expression must be double quotes.
- Backslashes are escape characters for WMI, so when you need to use a literal backslash, as in this example, you have to use two backslashes.
- The output of `Gwmi` always includes a number of system properties. These are often suppressed by PowerShell's default display configuration, but they'll be displayed if you're deliberately listing all properties or if the class doesn't have a default. System property names start with a double underscore. Here are two particularly useful ones:
  - `__SERVER` contains the name of the computer that the instance was retrieved from. This can be useful when retrieving WMI information from multiple computers at once.
  - `__PATH` is an absolute reference to the instance itself, and it can be used to requery the instance if necessary.

The cmdlet can retrieve not only from remote computers but from multiple computers, using any technique that can produce a collection of strings that contain either computer names or IP addresses, for example,

```
PS C:\> Gwmi Win32_BIOS -comp server-r2,server3,dc4
```

Computers are contacted sequentially, and if one computer isn't available, the cmdlet will produce an error, skip that computer, and move on to the next. Unavailable computers generally must time out, which means the cmdlet will pause for about 30–45 seconds until it gives up, produces the error, and moves on.

Once you retrieve a set of WMI instances, you can pipe them to any `-Object` cmdlet, to any `Format-` cmdlet, or to any of the `Out-`, `Export-`, or `ConvertTo-` cmdlets. For example, here's how you could produce a custom table from the Win32_BIOS class:

```
PS C:\> Gwmi Win32_BIOS | Format-Table SerialNumber,Version -auto
```

In chapter 8, I showed you a technique that can be used to produce custom columns using the `Format-Table` cmdlet. That technique can come in handy when you wish to query a couple of WMI classes from a given computer and have the results aggregated into a single table. To do so, you create a custom column for the table, and have that column's expression execute a whole new WMI query. The syntax for the command can be confusing, but the results are impressive:

```
PS C:\> gwmi -class win32_bios -computer server-r2,localhost | format-table
 @{l='ComputerName';e={$_.__SERVER}},@{l='BIOSSerial';e={$_.SerialNumber}},
@{l='OSBuild';e={gwmi -class win32_operatingsystem -comp $_.__SERVER | sele
ct-object -expand BuildNumber}} -autosize

ComputerName BIOSSerial                                                    OSBuil
                                                                           d
------------ ----------                                                    ------
SERVER-R2    VMware-56 4d 45 fc 13 92 de c3-93 5c 40 6b 47 bb 5b 86 7600
```

That syntax can be a bit easier to parse if you copy it into the PowerShell ISE and format it a bit:

```
gwmi -class win32_bios -computer server-r2,localhost |
 format-table
  @{l='ComputerName';e={$_.__SERVER}},
  @{l='BIOSSerial';e={$_.SerialNumber}},
  @{l='OSBuild';e={
    gwmi -class win32_operatingsystem -comp $_.__SERVER |
    select-object -expand BuildNumber}
  } -autosize
```

Here's what's happening:

- `Get-WmiObject` is querying Win32_BIOS from two computers.
- The results are being piped to `Format-Table`. `Format-Table` is being told to create three custom columns:
  - The first column is named ComputerName, and it's using the `__SERVER` system property from the `Win32_BIOS` instance.
  - The second column is named BIOSSerial, and it's using the `SerialNumber` property of the `Win32_BIOS` instance.
  - The third column is named OSBuild. This column is executing a whole new `Get-WmiObject` query, retrieving the `Win32_OperatingSystem` class from the `__SERVER` system property of the `Win32_BIOS` instance (of the same computer). That result is being piped to `Select-Object`, which is selecting just the contents of the `BuildNumber` property of the `Win32_OperatingSystem` instance and using that as the value for the OSBuild column.

That's complex syntax, but it offers powerful results. It's also a great example of how much you can achieve by stringing together a few carefully selected PowerShell cmdlets.

As I've mentioned, some WMI classes include methods. You'll see how to use those in chapter 13; doing so can be a bit complicated, and the topic deserves its own chapter.

## 11.6 WMI documentation

I mentioned earlier that a search engine is often the best way to find whatever WMI documentation exists. The `Win32_` classes are quite well documented in Microsoft's MSDN Library site, but a search engine remains the easiest way to land on the right page. I enter the name of the class, and the first hit in Google or Bing is usually a page on http://msdn.microsoft.com.

Figure 11.3 shows what a typical documentation page looks like.

Here are some tips for using these documentation pages:

- In the table of contents on the left side, click WMI Classes or Win32 Classes to go up a level to the full list of classes.
- In the main documentation page, methods (if any exist) are listed first, with links to each method's individual documentation page. Chapter 13 of this book will contain more information on interpreting the method documentation.
- Properties are listed next. Read these carefully! Sometimes Microsoft adds properties to a class in newer versions of Windows, meaning the property might not exist in older versions. Other times, different versions of Windows use properties differently, such as the `Win32_Processor` class. That class changed

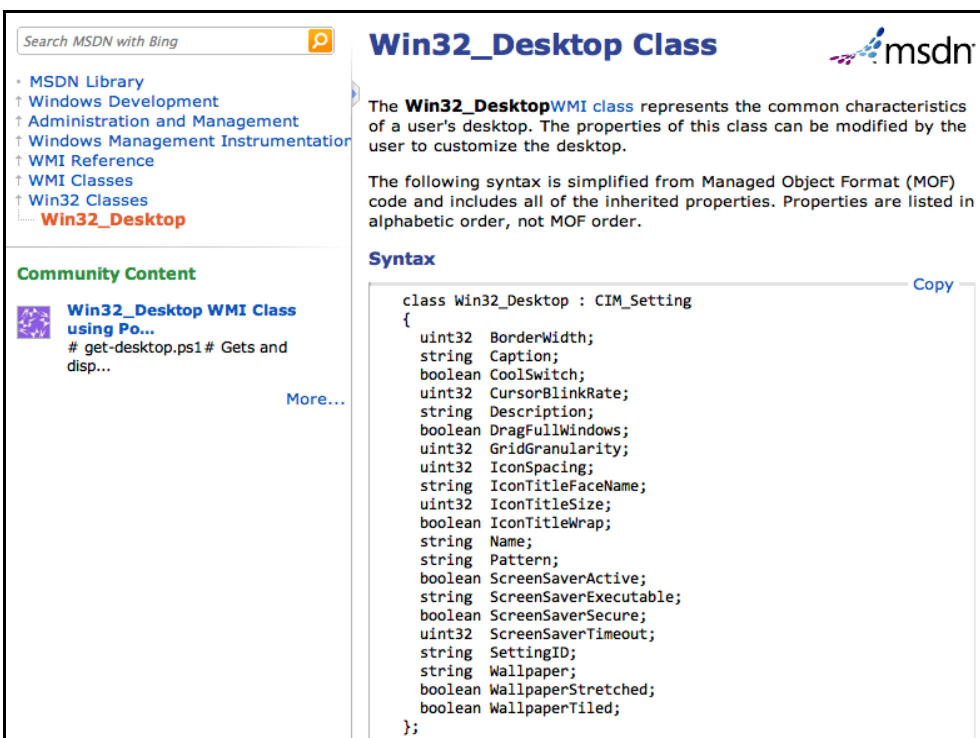



**Figure 11.3 Finding WMI documentation on the MSDN Library website**

between Windows XP and Windows Vista to streamline the way in which the class represents processor sockets and cores.

- At the end is a list of the operating systems that support the class. Newer versions of Windows contain more classes, so always check this list to make sure the class will exist on the version of Windows you need to use. Microsoft isn't always vigilant about keeping this list up to date, but it's a good guideline to start with.

## 11.7 Common points of confusion

Because I've spent the last ten chapters telling you to use the built-in PowerShell help, you might be inclined to run something like `help win32_service` right inside PowerShell. Sadly, that won't work. The operating system itself doesn't contain any WMI documentation, so PowerShell's help function wouldn't have anyplace to go look for it. You're stuck with whatever help you can find online—and much of that will be from other admins and programmers, not from Microsoft. Search for "root\SecurityCenter," for example, and you won't find a single Microsoft documentation page in the results, which is unfortunate.

The different filter criteria that WMI uses are also common points of confusion. You should always provide a filter whenever you need anything other than all of the available instances, but you'll have to memorize that different filter syntax. The filter syntax is passed along to WMI and not processed by PowerShell, which is why you have to use the syntax that WMI prefers instead of the native PowerShell operators.

Part of what makes WMI confusing for some of my classroom students is that, although PowerShell provides an easy way to query information from WMI, WMI isn't really integrated into PowerShell. WMI is an external technology, and it has its own rules and its own way of working. Although you can get to it from within PowerShell, it won't behave exactly like other cmdlets and techniques that are integrated completely within PowerShell. Keep that in mind, and watch for little points of confusion that result from WMI's individuality.

## 11.8 Lab

Take some time to complete the following hands-on tasks. Much of the difficulty in using WMI is in finding the class that will give you the information you need, so much of the time you'll spend in this lab will be tracking down the right class. Try to think in keywords (I'll provide some hints), and use a WMI explorer to quickly search through classes (the WMI explorer I use lists classes alphabetically, making it easier for me to validate my guesses).

1. What class could be used to view the current IP address of a network adapter? Does the class have any methods that could be used to release a DHCP lease? (Hint: *Network* is a good keyword here.)
2. Create a table that shows a computer name, operating system build number, operating system description (caption), and BIOS serial number. (Hint: You've

seen this technique, but you'll need to reverse it a bit and query the OS class first, then query the BIOS second).

3 Query a list of hotfixes using WMI. (Hint: Microsoft formally refers to these as *quick fix engineering*). Is the list different from that returned by the `Get-Hotfix` cmdlet?
4 Display a list of services, including their current status, their start mode, and the account they use to log on.
5 Can you find a class that will display a list of installed software *products*? Do you consider the resulting list to be complete?

## 11.9 Ideas for on your own

Think about some of the things you might want to query from WMI, and see if you can find the classes and properties that expose that information. For example, you might want to find the service pack version of a computer: a service pack modifies the operating system, so you might start your search in a class that has something to do with the operating system (and you've seen that class name in this chapter). Always try piping the object to `Gm` or `Format-List *` to see a full set of properties, or use a WMI explorer tool to review the full set of class properties.

# 12 Multitasking with background jobs

Everyone's always telling you to "multitask," right? Why shouldn't PowerShell help you out with that by doing more than one thing at a time? It turns out that PowerShell can do exactly that, especially for longer-running tasks that might involve multiple target computers. Make sure you've read chapters 10 and 11 before you dive into this chapter, because we're going to take those remoting and WMI concepts a bit further.

## 12.1 Making PowerShell do multiple things at the same time

You should think of PowerShell as a single-threaded application, meaning that it can only do one thing at once. You type a command, you hit Return, and the shell waits while that command executes. You can't start running a second command until the first finishes.

With its background jobs functionality, however, PowerShell has the ability to move a command onto a separate background thread (actually a separate, background PowerShell process). That enables the command to run in the background, while you continue using the shell for something else.

You have to make that decision before running the command; after you press Return, you can't decide to move a long-running command into the background. After commands are in the background, PowerShell provides mechanisms to check on their status, retrieve any results, and so forth.

## 12.2 Synchronous versus asynchronous

Let's get a few bits of terminology out of the way first. PowerShell runs normal commands *synchronously*, meaning you hit Return and then wait for the command to complete. Moving a job into the background allows it to run *asynchronously*, meaning that you can continue using the shell for other tasks while the command completes.

There are a few important differences between running commands in these two ways:

- When you run a command synchronously, you can respond to input requests. When running commands in the background, there's no opportunity to see input requests—in fact, they'll stop the command from running.
- Synchronous commands produce error messages when something goes wrong. Background commands produce errors, but you won't see them immediately. You'll have to make arrangements to capture them, if necessary. (Chapter 22 discusses that.)
- If you omit a required parameter on a synchronous command, PowerShell can prompt you for the missing information. On a background command, it can't, so the command will simply fail.
- The results of a synchronous command start displaying as soon as results become available. With a background command, you wait until the command finishes running and then retrieve the cached results.

I typically run commands synchronously to test them out and get them working properly, and only run them in the background after I know they're fully debugged and working as I expect them to. That way, I can ensure that the commands will run without problems, and that they will have the best chance of completing in the background.

PowerShell refers to background commands as *jobs*, and there are several ways to create jobs, along with several commands to manage them.

> **Above and beyond**
>
> Technically, the jobs that I'll discuss in this chapter are just one kind of jobs that you can encounter. Jobs are an extension point for PowerShell, meaning that it's possible for someone (either in Microsoft or as a third party) to create other things called jobs that look and work a bit differently than what I'll describe here. I just wanted you to know that little detail, and to know that what you'll learn here only applies to the native jobs that ship with PowerShell v2.

## 12.3 Creating a local job

The first type of job we'll cover is perhaps the easiest: a local job. This is a command that runs more or less entirely on your local computer (with exceptions that I'll cover in a second) and that runs in the background.

To launch one of these jobs, you use the `Start-Job` command. A `-scriptblock` parameter lets you specify the command (or commands) to run. PowerShell will make up a default job name (Job1, Job2, and so on), or you can specify a custom job name by using the `-Name` parameter. If you need the job to run under alternative credentials, a `-credential` parameter will accept a DOMAIN\Username credential and prompt you for the password. Rather than specifying a script block, you can specify the `-FilePath` parameter to have the job execute an entire script file full of commands.

Here's a simple example:

```
PS C:\> start-job -scriptblock { dir }

Id          Name         State        HasMoreData      Location
--          ----         -----        -----------      --------
1           Job1         Running      True             localhost
```

The result of the command is the job object that was created, and you can see that the job immediately begins running. The job is also assigned a sequential job ID number, which is shown in the table.

I said that these jobs run entirely on your local computer, and that's basically true. But the commands in the job are allowed to access remote computers, which would be the case if you ran a command that supported a `-computerName` parameter. Here's an example:

```
PS C:\> start-job -scriptblock {
➥ get-eventlog security -computer server-r2
}

Id          Name         State        HasMoreData      Location
--          ----         -----        -----------      --------
3           Job3         Running      True             localhost
```

> **TRY IT NOW** Hopefully you'll follow along and run all of these commands. If you only have a single computer to work with, refer to its computer name and use localhost as an alternative, so that PowerShell will act like it's dealing with two computers.

The processing for this job will happen on your local computer. It will contact the specified remote computer (SERVER-R2 in this example), so the job is, in a way, a "remote job." But because the command itself is running locally, I still refer to this as a local job.

Sharp-eyed readers will note that the first job I created was named Job1 and given the ID 1, but the second job was Job3 with ID 3. It turns out that every job has at least one *child job*, and the first child job (a child of Job1) was given the name Job2 and the ID 2. We'll get to child jobs a bit later in this chapter.

Here's something to keep in mind: although local jobs run entirely locally, they do require the infrastructure of PowerShell's remoting system, which we covered in chapter 10. If you haven't enabled remoting, you won't be able to start local jobs.

## 12.4 WMI, as a job

Another way to start a job is to use `Get-WmiObject`. As I explained in the previous chapter, that command can contact one or more remote computers, but it does so sequentially. That means a long list of computer names can cause the command to take a long time to process, and it's a natural choice for moving to a background job. To do so, you use `Get-WmiObject` as normal but add the `-AsJob` parameter. You don't get to specify a custom job name here; you're stuck with the default job name that PowerShell makes up.

> **TRY IT NOW** If you're running the same commands on your test system, you'll need to create a text file called allservers.txt. I put it in the root of my C: drive (because that's where I have PowerShell focused for these examples), and I put several computer names in the file, listing one name per line. You can list your computer name and localhost to duplicate the results I'm showing you.

```
PS C:\> get-wmiobject win32_operatingsystem -computername
➥ (get-content allservers.txt) -asjob

WARNING: column "Command" does not fit into the display and was removed.

Id              Name            State         HasMoreData     Location
--              ----            -----         -----------     --------
5               Job5            Running       False           server-r2,lo...
```

This time, the shell will create one top-level parent job (Job5, which is shown in the output of the command), and it will create one child job for each computer that you specified. You can see that the Location column in the output table lists as many of the computer names as will fit, indicating that the job is going to be running against those computers.

It's important to understand that `Get-WmiObject` is executing only on your computer; the cmdlet is using normal WMI communications to contact the remote computers that you specified. It will still do so one at a time and follow the usual defaults of skipping computers that aren't available, and so forth. In fact, it works identically to using `Get-WmiObject` synchronously, except that the cmdlet runs in the background.

> **TRY IT NOW** There are a few commands other than `Get-WmiObject` that can start a job. Try running `Help * -parameter asjob` to see if you can find them all.

## 12.5 Remoting, as a job

The last way to create a new job is to use PowerShell's remoting capabilities, which you learned about in chapter 10. As with `Get-WmiObject`, you start this kind of job by adding an `-AsJob` parameter, but this time you'll add it to the `Invoke-Command` cmdlet.

There's an important difference here: whatever command you specify in the `-scriptblock` (or `-command`, which is an alias for the same parameter) will be transmitted in parallel to each computer that you specified. Up to 32 computers will be contacted at once (unless you modify the `-throttleLimit` parameter to allow more or fewer), so if you specify more than 32 computer names, only the first 32 will start. The rest will start after the first set begins finishing, and the top-level job will show a completed status after all of the computers are finished.

Unlike the other two ways of starting a job, this one requires that PowerShell v2 be installed on each target computer, and that each target computer have PowerShell remoting enabled. Because the command physically executes on each remote computer, you're distributing the computing workload, which can help improve performance for complex or long-running commands. The results come back to your computer and are stored with the job until you're ready to review them.

Here's an example, where you'll also see the `-JobName` parameter that lets you specify a job name other than the boring default:

```
PS C:\> invoke-command -command { get-process }
➥ -computername (get-content .\allservers.txt )
➥ -asjob -jobname MyRemoteJob

WARNING: column "Command" does not fit into the display and was removed.

Id            Name            State         HasMoreData      Location
--            ----            -----         -----------      --------
8             MyRemoteJob     Running       True             server-r2,lo...
```

## 12.6 Getting job results

The first thing you'll probably want to do is check to see if your jobs have finished. The `Get-Job` cmdlet will retrieve every job currently defined by the system, and show you each one's status:

```
PS C:\> get-job

Id            Name            State         HasMoreData      Location
--            ----            -----         -----------      --------
1             Job1            Completed     True             localhost
3             Job3            Completed     True             localhost
5             Job5            Completed     True             server-r2,lo...
8             MyRemoteJob     Completed     True             server-r2,lo...
```

You can also retrieve a specific job, either by using its ID or its name. I suggest that you do that and pipe the results to `Format-List *`, because you've gathered some valuable information:

```
PS C:\> get-job -id 1 | format-list *

State         : Completed
HasMoreData   : True
StatusMessage :
Location      : localhost
Command       :  dir
JobStateInfo  : Completed
Finished      : System.Threading.ManualResetEvent
InstanceId    : e1ddde9e-81e7-4b18-93c4-4c1d2a5c372c
Id            : 1
Name          : Job1
ChildJobs     : {Job2}
Output        : {}
Error         : {}
```

```
Progress       : {}
Verbose        : {}
Debug          : {}
Warning        : {}
```

> **TRY IT NOW** If you're following along, keep in mind that your job IDs and names might be a bit different than mine. Focus on the output of `Get-Job` to get your job IDs and names, and substitute yours in the examples.

One of the most important pieces of information there is the `ChildJobs` property, which we'll cover in just a moment.

To retrieve the results from a job, use `Receive-Job`. Before you run this, you need to know a few things:

- You have to specify the job you want to receive results from. You can do this by job ID, job name, or by getting jobs with `Get-Job` and piping them to `Receive-Job`.
- If you receive the results of the parent job, those results will include all output from all child jobs. Alternatively, you can choose to just get the results from one or more child jobs.
- Normally, receiving the results from a job clears them out of the job output cache, so you can't get them a second time. Specify `-keep` to keep a copy of the results in memory. Or, you can output the results to CliXML if you want to retain a copy to work with.
- The job results may be deserialized objects, which you learned about in chapter 10. That means they're a snapshot from the point in time when they were generated, and they may not have any methods that you can execute. But you can pipe the job results directly to cmdlets such as `Sort-Object`, `Format-List`, `Export-CSV`, `ConvertTo-HTML`, `Out-File`, and so on, if desired.

Here's an example:

```
PS C:\> receive-job -id 1

    Directory: C:\Users\Administrator\Documents

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        11/21/2009  11:53 AM            Integration Services Script Component
d----        11/21/2009  11:53 AM            Integration Services Script Task
d----         4/23/2010   7:54 AM            SQL Server Management Studio
d----         4/23/2010   7:55 AM            Visual Studio 2005
d----        11/21/2009  11:50 AM            Visual Studio 2008
```

This is an interesting set of results. Here's a quick reminder of the command that launched this job in the first place:

```
PS C:\> start-job -scriptblock { dir }
```

Although my shell was in the C:\ drive when I ran this, the directory in the results is C:\Users\Administrator\Documents. As you can see, even local jobs take on a slightly different context when they run, which may result in a change of location. Don't ever make assumptions about file paths from within a background job: use absolute paths to make sure you can refer to whatever files your job command may require. If I wanted the background job to get a directory of C:\, I should have run this:

```
PS C:\> start-job -scriptblock { dir c:\ }
```

When I received the results from Job1, I didn't specify `-keep`. If I try to get those same results again, I'll get nothing, because the results are no longer cached with the job:

```
PS C:\> receive-job -id 1
PS C:\>
```

Here's how you would force the results to stay cached in memory:

```
PS C:\> receive-job -id 3 -keep

Index  Time          EntryType    Source                 InstanceID  Message
-----  ----          ---------    ------                 ----------  -------
6542   Oct 04 11:55  SuccessA...  Microsoft-Windows...         4634  An...
6541   Oct 04 11:55  SuccessA...  Microsoft-Windows...         4624  An...
6540   Oct 04 11:55  SuccessA...  Microsoft-Windows...         4672  Sp...
6539   Oct 04 11:54  SuccessA...  Microsoft-Windows...         4634  An...
```

Of course, you'll eventually want to free up the memory that's being used to cache the job results, and I'll cover that in a bit. But first, let's see a quick example of piping the job results directly to another cmdlet:

```
PS C:\> receive-job -name myremotejob | sort-object PSComputerName |
➥ Format-Table -groupby PSComputerName

   PSComputerName: localhost

Handles  NPM(K)  PM(K)  WS(K) VM(M)  CPU(s)    Id  ProcessName PSComputerName
-------  ------  -----  ----- -----  ------    --  ----------- --------------
    195      10   2780   5692    30    0.70   484  lsm         loca...
    237      38  40704  36920   547    3.17  1244  Micro...    loca...
    146      17   3260   7192    60    0.20  3492  msdtc       loca...
   1318     100  42004  28896   154   15.31   476  lsass       loca...
```

This was the job that I started by using `Invoke-Command`. As always, the cmdlet has added the `PSComputerName` property so that I can keep track of which object came from which computer. Because I retrieved the results from the top-level job, this included all of the computers that I specified, so this command will sort them on the computer name and then create an individual table group for each computer.

Get-Job can keep you informed about which jobs have results remaining:

```
PS C:\> get-job

WARNING: column "Command" does not fit into the display and was removed.

Id              Name            State        HasMoreData     Location
--              ----            -----        -----------     --------
1               Job1            Completed    False           localhost
3               Job3            Completed    True            localhost
5               Job5            Completed    True            server-r2,lo...
8               MyRemoteJob     Completed    False           server-r2,lo...
```

The HasMoreData column will be False when there is no output cached with that job. In the case of Job1 and MyRemoteJob, it's because I already received those results and didn't specify -keep when I did so.

## 12.7 Working with child jobs

I mentioned earlier that all jobs consist of one top-level parent job and at least one child job. Let's look at a job again:

```
PS C:\> get-job -id 1 | format-list *

State         : Completed
HasMoreData   : True
StatusMessage :
Location      : localhost
Command       :  dir
JobStateInfo  : Completed
Finished      : System.Threading.ManualResetEvent
InstanceId    : e1ddde9e-81e7-4b18-93c4-4c1d2a5c372c
Id            : 1
Name          : Job1
ChildJobs     : {Job2}
Output        : {}
Error         : {}
Progress      : {}
Verbose       : {}
Debug         : {}
Warning       : {}
```

**TRY IT NOW** Don't follow along for this part, because if you've been following along up to now, you've already received the results of Job1. If you'd like to try this, start a new job by running Start-Job -script { Get-Service }, and use that new job's ID instead of the ID number 1 I use in my example.

Here, you can see that Job1 has a child job, Job2. You can get it directly now that you know its name:

```
PS C:\> get-job -name job2 | format-list *

State         : Completed
StatusMessage :
HasMoreData   : True
Location      : localhost
Runspace      : System.Management.Automation.RemoteRunspace
Command       :  dir
JobStateInfo  : Completed
Finished      : System.Threading.ManualResetEvent
InstanceId    : a21a91e7-549b-4be6-979d-2a896683313c
Id            : 2
Name          : Job2
ChildJobs     : {}
Output        : {Integration Services Script Component, Integration Servic
                es Script Task, SQL Server Management Studio, Visual Studi
                o 2005...}
Error         : {}
Progress      : {}
Verbose       : {}
Debug         : {}
Warning       : {}
```

Sometimes, a job will have too many child jobs to list in that form, so you may want to list them a bit differently:

```
PS C:\> get-job -id 1 | select-object -expand childjobs

WARNING: column "Command" does not fit into the display and was removed.

Id            Name          State         HasMoreData     Location
--            ----          -----         -----------     --------
2             Job2          Completed     True            localhost
```

That technique will create a table of the child jobs for job ID 1, and the table can obviously be however long it needs to be to list them all.

You can receive the results from any individual child job—specify its name or ID with `Receive-Job`.

## 12.8 Commands for managing jobs

There are three more commands used with jobs. For each of these, you may specify a job either by giving its ID, giving its name, or by getting the job and piping it to one of these cmdlets:

- `Remove-Job`—This deletes a job, and any output still cached with it, from memory.
- `Stop-Job`—If a job seems to be stuck, this command will terminate it. You'll still be able to receive whatever results were generated to that point.

- `Wait-Job`—This is useful if a script is going to start a job and you want the script to continue only when the job is done. This command forces the shell to stop and wait until the job is completed, and then allows the shell to continue.

For example, to remove the jobs that I've already received output from, I'd use this command:

```
PS C:\> get-job | where { -not $_.HasMoreData } | remove-job
PS C:\> get-job

WARNING: column "Command" does not fit into the display and was removed.

Id             Name           State          HasMoreData       Location
--             ----           -----          -----------       --------
3              Job3           Completed      True              localhost
5              Job5           Completed      True              server-r2,lo...
```

Jobs can also fail, meaning that something went wrong with their execution. Consider this example:

```
PS C:\> invoke-command -command { nothing } -computer notonline -asjob -job
name ThisWillFail

WARNING: column "Command" does not fit into the display and was removed.

Id             Name             State       HasMoreData       Location
--             ----             -----       -----------       --------
11             ThisWillFail     Failed      False             notonline
```

Here, I started a job with a bogus command, and targeted a nonexistent computer. The job immediately failed, as shown in its status. I don't need to use `Stop-Job` here; the job isn't running. I can, however, get a list of its child jobs:

```
PS C:\> get-job -id 11 | format-list *

State          : Failed
HasMoreData    : False
StatusMessage  :
Location       : notonline
Command        :  nothing
JobStateInfo   : Failed
Finished       : System.Threading.ManualResetEvent
InstanceId     : d5f47bf7-53db-458d-8a08-07969305820e
Id             : 11
Name           : ThisWillFail
ChildJobs      : {Job12}
Output         : {}
Error          : {}
Progress       : {}
Verbose        : {}
Debug          : {}
Warning        : {}
```

And I can then get just that child job:

```
PS C:\> get-job -name job12

WARNING: column "Command" does not fit into the display and was removed.

Id              Name            State         HasMoreData        Location
--              ----            -----         -----------        --------
12              Job12           Failed        False              notonline
```

As you can see, there are no results to retrieve because no output was ever created for this job. But the job's errors are stored in the results, and you can get them by using `Receive-Job`:

```
PS C:\> receive-job -name job12
Receive-Job : [notonline] Connecting to remote server failed with the foll
owing error message : WinRM cannot process the request. The following erro
r occured while using Kerberos authentication: The network path was not fo
und.
```

The actual error is much longer; I've truncated it here to save some space. You'll notice that the error includes the computer name that the error came from, `[notonline]`. What happens if only one of the computers can't be reached? Let's try:

```
PS C:\> invoke-command -command { nothing }
➥ -computer notonline,server-r2 -asjob -jobname ThisWillFail

WARNING: column "Command" does not fit into the display and was removed.

Id              Name              State        HasMoreData       Location
--              ----              -----        -----------       --------
13              ThisWillFail      Running      True              notonline,se...
```

After waiting for a bit, I'll run this:

```
PS C:\> get-job

WARNING: column "Command" does not fit into the display and was removed.

Id              Name              State        HasMoreData       Location
--              ----              -----        -----------       --------
13              ThisWillFail      Failed       False             notonline,se...
```

The job still failed, but let's look at the individual child jobs:

```
PS C:\> get-job -id 13 | select -expand childjobs

WARNING: column "Command" does not fit into the display and was removed.

Id              Name              State        HasMoreData       Location
--              ----              -----        -----------       --------
14              Job14             Failed       False             notonline
15              Job15             Failed       False             server-r2
```

Okay, they both failed. I have a feeling I know why Job14 didn't work, but what's wrong with Job15?

```
PS C:\> receive-job -name job15
Receive-Job : The term 'nothing' is not recognized as the name of a cmdlet
, function, script file, or operable program. Check the spelling of the na
me, or if a path was included, verify that the path is correct and try aga
in.
```

Ah, that's right, I told it to run a bogus command. As you can see, each child job can fail for different reasons, and PowerShell will track each one individually.

## 12.9 Common points of confusion

Jobs are usually pretty straightforward, but there's one thing I've seen folks do that does cause confusion. Don't do this:

```
PS C:\> invoke-command -command { Start-Job -scriptblock { dir } }
➥ -computername Server-R2
```

This is starting up a temporary connection to SERVER-R2 and starting a local job. Unfortunately, that connection immediately terminates, so there's no way to reconnect and retrieve that job. In general, then, don't mix and match the three ways of starting jobs.

This one is also a bad idea:

```
PS C:\> start-job -scriptblock { invoke-command -command { dir }
➥ -computername SERVER-R2 }
```

That's completely redundant; just keep the `Invoke-Command` part and use the `-AsJob` parameter to have it run in the background.

Less confusing, but equally interesting, are the questions my classroom students often ask about jobs. Probably the most important of these is, "Can I see jobs started by someone else?" The answer is, no. Jobs are contained entirely within the PowerShell process, and although you could see that another user was running PowerShell, you wouldn't be able to see inside that process. It's just like any other application: you could see that another user was running Microsoft Office Word, for example, but you couldn't see what documents they were editing, because those documents exist entirely inside of Word's process.

Jobs only last as long as your PowerShell session is open. After you close it, any jobs defined within it are gone. Jobs aren't defined anywhere outside of PowerShell, so they depend upon its process continuing to run in order to maintain themselves.

# 13 Working with bunches of objects, one at a time

Pretty much the whole point of PowerShell is to automate administration, and that often means you'll want to perform some tasks with multiple targets. You might want to reboot several computers, reconfigure several services, modify several mailboxes, and so on. In this chapter, you'll learn three distinct techniques for accomplishing these and other multiple-target tasks: batch cmdlets, WMI methods, and object enumeration.

## 13.1 Automation for mass management

I know that this isn't a book about VBScript, but I want to use a VBScript example to briefly illustrate the way that multiple-target administration—what I like to call *mass management*—has been approached in the past. Consider this example (there's no need to type this in and run it—we're just going to discuss the approach, not the results):

```
For Each varService in colServices
  varService.ChangeStartMode("Automatic")
Next
```

This kind of approach isn't common only in VBScript, but is common throughout the world of programming. Here's what it does:

1. Assume that the variable `colServices` contains multiple services. It doesn't matter how they got in there, because there are many ways you could retrieve the services. What matters right now is that you have already retrieved the services and put them into this variable.

2. The `For Each` construct will go through, or *enumerate*, the services one at a time. As it does so, it will place each service into the variable `varService`. So, within the construct, `varService` will only contain a single service. If `colServices` contained 50 services, then the construct's contents would execute 50 times, and each time, `varService` would contain a different one of the 50 services.
3. Within the construct, we're executing a method—in this example, `ChangeStartMode()`—to perform some task.

If you think about it carefully, you'll realize that we really aren't doing something to a bunch of services at once. Instead, we're doing something to one service at a time, exactly as we would if we were manually reconfiguring the services by using the graphical user interface. The only difference is that we're making the computer go through the services one at a time.

Computers are really good at repeating things over and over, so this isn't a horrible approach. The problem is that this approach requires us to give the computer a longer and fairly complicated set of instructions. Learning the language necessary to give that set of instructions can take a while, which is why a lot of administrators try to avoid VBScript and other scripting languages.

PowerShell can duplicate this approach, and I'll show you how later in this chapter, because sometimes you have to resort to this method. But the approach of having the computer enumerate objects isn't the most efficient way to use PowerShell. In fact, PowerShell offers two other techniques that are easier to learn and easier to type, and they're often more powerful.

## 13.2 The preferred way: batch cmdlets

As you've learned in several previous chapters, many PowerShell cmdlets can accept batches, or *collections*, of objects to work with.

In chapter 7, for example, you learned how objects can be piped from one cmdlet to another, like this (please don't actually run this—it'll probably crash your computer):

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

This is an example of batch administration using a cmdlet. In this case, `Stop-Service` is specifically designed to accept one service object, or many service objects, from the pipeline, and then stop them. `Set-Service`, `Stop-Process`, `Move-ADObject`, and `Move-Mailbox` are all examples of cmdlets that accept one or more input objects and then perform some task or action with each of them. You don't need to manually enumerate the objects using a construct, as I did in the VBScript example in the previous section. PowerShell knows how to work with batches of objects, and can handle them for you with a less-complex syntax.

These so-called *batch cmdlets* (that's my name for them, not an official term) are my preferred way of performing mass management. For example, let's suppose I need to change the start mode of three services. Rather than using an approach like the VBScript one, I could do this:

```
Get-Service -name BITS,Spooler,W32Time | Set-Service -startuptype Automatic
```

In a way, `Get-Service` is also a kind of batch cmdlet, because it's capable of retrieving services from multiple computers. Suppose I needed to change those same three services across a set of three computers:

```
Get-Service -name BITS,Spooler,W32Time -computer Server1,Server2,Server3 |
➥ Set-Service -startuptype Automatic
```

One potential downside of this approach is that cmdlets that perform an action often don't produce any output indicating that they've done their job. That means there is no visual output from either of the preceding commands, which can be disconcerting. But those cmdlets often have a `-passThru` parameter, which tells them to output whatever objects they accepted as input. I could have `Set-Service` output the same services it just modified, and have `Get-Service` re-retrieve those services to see if the change took effect.

Here's an example of using `-passThru` with a different cmdlet:

```
Get-Service -name BITS -computer Server1,Server2,Server3 |
➥ Start-Service -passthru |
➥ Get-Service
```

This command would retrieve the specified service from the three computers I listed. The services would be piped to `Start-Service`, which would not only start them but also output the original service objects. Those service objects would be piped to `Get-Service`, telling it which services to retrieve. It would then re-retrieve the services and create the usual output, enabling me to see that the services were started successfully.

Once more: this is the preferred way to work in PowerShell. If a cmdlet exists to do whatever you want, you should use it. Ideally, cmdlets are always written to work with batches of objects. That isn't always the case (cmdlet authors are still learning the best ways to write cmdlets for us administrators), but it's the ideal.

## 13.3 The WMI way: invoking WMI methods

Unfortunately, we don't always have cmdlets that can take whatever action we need, and that's especially true when it comes to the items we can manipulate through Windows Management Instrumentation (WMI, which we tackled in chapter 11).

For example, consider the `Win32_NetworkAdapterConfiguration` class in WMI. This class represents the configuration bound to a network adapter (adapters can have multiple configurations, but for now let's assume they only have one configuration apiece, which is common on client computers). Let's say that my goal is to enable DHCP on all of my computer's Intel network adapters—I don't want any of the RAS or other virtual adapters.

I might start by trying to query the desired adapter configurations, so that I get something like this as output:

```
DHCPEnabled      : False
IPAddress        : {192.168.10.10, fe80::ec31:bd61:d42b:66f}
DefaultIPGateway :
DNSDomain        :
ServiceName      : E1G60
Description      : Intel(R) PRO/1000 MT Network Connection
Index            : 7

DHCPEnabled      : True
IPAddress        :
DefaultIPGateway :
DNSDomain        :
ServiceName      : E1G60
Description      : Intel(R) PRO/1000 MT Network Connection
Index            : 12
```

To achieve this output, I would need to query the appropriate WMI class and filter it so that only configurations with "Intel" in their description were included. Here's the command that will do it (notice that the `%` acts as a wildcard within the WMI filter syntax):

```
PS C:\> gwmi win32_networkadapterconfiguration
➥ -filter "description like '%intel%'"
```

> **TRY IT NOW** You're welcome to follow along with the commands I'm running in this section of the chapter. You may need to tweak the commands slightly to make them work. For example, if your computer doesn't have any Intel-made network adapters, you'd need to change the filter criteria appropriately.

Once I have those configuration objects in the pipeline, I want to enable DHCP on them (you can see that one of my adapters doesn't have DHCP enabled). So I might start looking for a cmdlet named something like "Enable-DHCP." Unfortunately, I won't find it, because there's no such thing. There aren't any cmdlets that are capable of dealing directly with WMI objects in batches.

My next step would be to see if the object itself has a method that's capable of enabling DHCP. To find out, I'll pipe those configuration objects to `Get-Member` (or its alias, `Gm`):

```
PS C:\> gwmi win32_networkadapterconfiguration
➥ -filter "description like '%intel%'" | gm
```

Right near the top of the resulting list, I should see the method that I'm after: `EnableDHCP()`:

```
TypeName: System.Management.ManagementObject#root\cimv2\Win32_NetworkAd
apterConfiguration

Name                        MemberType      Definition
----                        ----------      ----------
DisableIPSec                Method          System.Management.ManagementB...
EnableDHCP                  Method          System.Management.ManagementB...
EnableIPSec                 Method          System.Management.ManagementB...
EnableStatic                Method          System.Management.ManagementB...
```

The next step a lot of PowerShell newcomers try is to pipe the configuration objects to the method:

```
PS C:\> gwmi win32_networkadapterconfiguration
➥ -filter "description like '%intel%'" | EnableDHCP()
```

Sadly, that won't work. You can't pipe objects to a method; you can only pipe to a cmdlet. `EnableDHCP` isn't actually a PowerShell cmdlet. Rather, it's an action that's directly attached to the configuration object itself. The old, VBScript-style approach would look a lot like the VBScript example I showed you at the start of this chapter, but with PowerShell you can do something simpler.

Although there's no "batch" cmdlet called Enable-DHCP, there is a generic cmdlet called `Invoke-WmiMethod`. This cmdlet is specially designed to accept a batch of WMI objects, such as my `Win32_NetworkAdapterConfiguration` objects, and to invoke one of the methods attached to those objects. So here's the command I would run:

```
PS C:\> gwmi win32_networkadapterconfiguration
➥ -filter "description like '%intel%'" |
➥ Invoke-WmiMethod -name EnableDHCP
```

There are a few things to keep in mind:

- The method name isn't followed by parentheses
- The method name isn't case-sensitive
- `Invoke-WmiMethod` can only accept one kind of WMI object at a time. In this case, I'm only sending it `Win32_NetworkAdapterConfiguration` objects, so it will work fine. It's okay to send it more than one object (that's the whole point, in fact), but all of the objects have to be of the same type.

The output of `Invoke-WmiMethod` can be a little confusing. WMI always produces a result object, and it has a lot of system properties (whose names start with two underscore characters). In my case, the command produced this:

```
__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
```

```
__DERIVATION      : {}
__SERVER          :
__NAMESPACE       :
__PATH            :
ReturnValue       : 0

__GENUS           : 2
__CLASS           : __PARAMETERS
__SUPERCLASS      :
__DYNASTY         : __PARAMETERS
__RELPATH         :
__PROPERTY_COUNT  : 1
__DERIVATION      : {}
__SERVER          :
__NAMESPACE       :
__PATH            :
ReturnValue       : 84
```

The only useful information here is the one property that doesn't start with two underscores: `ReturnValue`. That number tells me the result of the operation. A Google search for "Win32_NetworkAdapterConfiguration" turns up the documentation page, and I can then click through to the `EnableDHCP` method to see the possible return values and what they mean. Figure 13.1 shows what I discovered.

Zero appears to mean success, while 84 says that IP isn't enabled on that adapter configuration, so DHCP can't be enabled. But which bit of the output went with which of my two network adapter configurations? It's difficult to tell, because the output

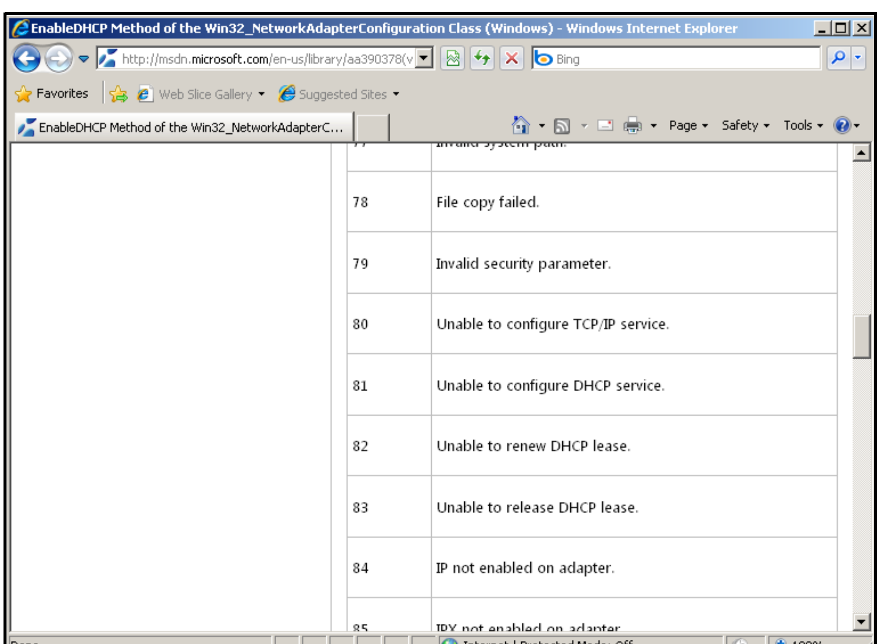



**Figure 13.1 Looking up return values for a WMI method's results**

doesn't tell you which specific configuration object produced it. That's unfortunate, but it's the way WMI works.

`Invoke-WmiMethod` will work for most situations where you have a WMI object that has a method that you want to execute. It works great when querying WMI objects from remote computers too. My basic rule is, "If you can get to something by using `Get-WmiObject`, then `Invoke-WmiObject` can execute its methods."

## 13.4 The backup plan: enumerating objects

Unfortunately, I have run across a few situations where `Invoke-WmiObject` couldn't execute a method—it kept returning weird error messages. I've also run into cases where I have a cmdlet that can produce objects, but there is no batch cmdlet to which I can pipe those objects to take some kind of action. In either case, you can still perform whatever task you wanted to perform, but you'll have to fall back on the old VBScript-style approach of instructing the computer to enumerate the objects and perform your task against one object at a time. There are two ways to accomplish this in PowerShell: one is using a cmdlet, and the other is using a scripting construct. We'll focus on the first way in this chapter, and I'll save the second way for chapter 21, which dives into PowerShell's built-in scripting language.

As an example of how to do this, I'm going to use the `Win32_Service` WMI class. Specifically, I'm going to use the `Change()` method. This is a complex method that can change several elements of a service at once. Figure 13.2 shows its online

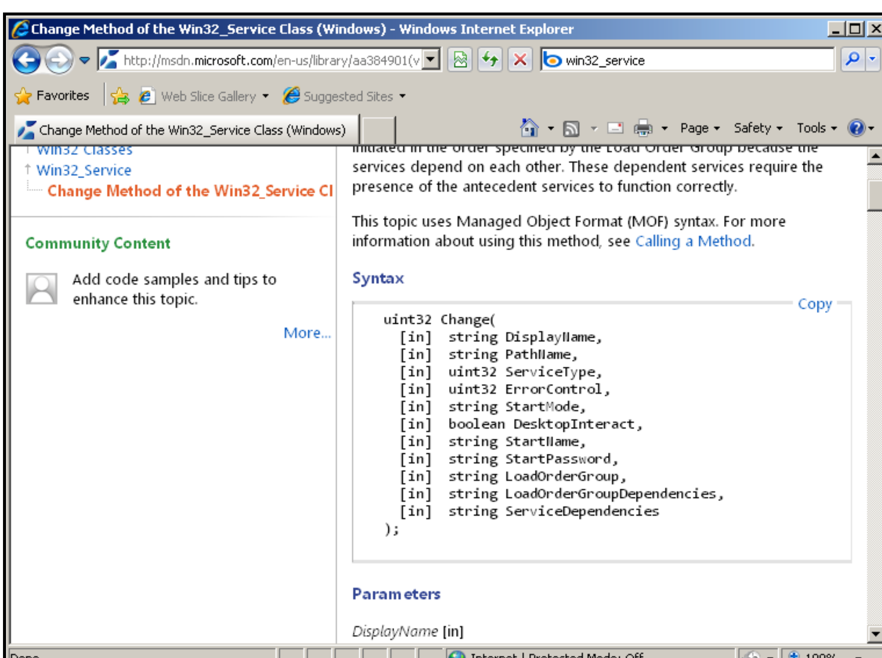



**Figure 13.2 Documentation page for the `Change()` method of `Win32_Service`**

documentation (which I found by searching for "Win32_Service," and then clicking on the `Change` method).

Reading this page, I discover that I don't have to specify every single parameter of the method. I can specify `Null` (which in PowerShell is in the special built-in `$null` variable) for any parameters that I want to omit. I want to change the service's startup password, which is the eighth parameter, so I'll need to specify `$null` for the first seven parameters. That means my method execution might look something like this:

```
Change($null, $null, $null, $null, $null, $null, $null, "P@ssw0rd")
```

By the way, the reason I'm not using `Get-Service` and `Set-Service` is that those cmdlets are incapable of displaying or setting a service's logon password. WMI can do it, though, so I'm using WMI.

Because I can't use the `Set-Service` batch cmdlet, which would normally be my preferred approach, I'll try my second approach, which is to use `Invoke-WmiMethod`. The cmdlet has a parameter, `-ArgumentList`, where I can specify the arguments for the method. Here's what I try, along with the result I get:

```
PS C:\> gwmi win32_service -filter "name = 'BITS'" | invoke-wmimethod -name
 change -arg $null,$null,$null,$null,$null,$null,$null,"P@ssw0rd"
Invoke-WmiMethod : Input string was not in a correct format.
At line:1 char:62
+ gwmi win32_service -filter "name = 'BITS'" | invoke-wmimethod <<<<  -nam
e change -arg $null,$null,$null,$null,$null,$null,$null,"P@ssw0rd"
    + CategoryInfo          : NotSpecified: (:) [Invoke-WmiMethod], Forma
   tException
    + FullyQualifiedErrorId : System.FormatException,Microsoft.PowerShell
   .Commands.InvokeWmiMethod
```

At this point, I have to make a decision. It's possible that I'm running the command incorrectly, so I have to decide if I want to spend a lot of time figuring it out. It's also possible that `Invoke-WmiMethod` just doesn't work with the `Change()` method very well, in which case I could be spending a lot of time trying to fix something that I have no control over.

My choice in these situations is to try a different approach: I'm going to ask the computer (well, the shell) to enumerate the service objects, one at a time, and execute the `Change()` method on each of them, one at a time. To do so, I'll use the `ForEach-Object` cmdlet:

```
PS C:\> gwmi win32_service -filter "name = 'BITS'" | foreach-object {$_.cha
nge($null,$null,$null,$null,$null,$null,$null,"P@ssw0rd") }

__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
```

```
__PATH           :
ReturnValue      : 0
```

The documentation page says that a `ReturnValue` of `0` means success, so that means I achieve my task. But let's look at that command in more detail, with a bit nicer formatting:

```
Get-WmiObject Win32_Service -filter "name = 'BITS'" |
➥ ForEach-Object -process {
➥ $_.change($null,$null,$null,$null,$null,$null,$null,"P@ssw0rd")
➥ }
```

There's a lot going on there. The first line should make sense: I'm using `Get-WmiObject` to retrieve all instances of `Win32_Service` that match my filter criteria, which is looking for services that have the name "BITS" (as usual, I'm picking on the BITS service because it's less essential than some others I could have picked, so breaking it won't crash my computer). I'm piping those `Win32_Service` objects to the `ForEach-Object` cmdlet.

Let's break that down into its component elements:

- First, there's the cmdlet name: `ForEach-Object`.
- Next, I'm using the `-Process` parameter to specify a script block. I didn't originally type the `-Process` parameter name, because it's a positional parameter. But that script block—everything contained within the curly braces—is the value for the `-Process` parameter. I went ahead and included the parameter name when I reformatted the command for easier reading.
- `ForEach-Object` will execute its script block once for each object that was piped into `ForEach-Object`. Each time the script block executes, the next piped-in object will be placed into the special `$_` placeholder.
- By following `$_` with a period, I'm telling the shell that I want to access a property or method of the current object.
- In this example, I'm accessing the `Change()` method. Note that the method's parameters are passed as a comma-separated list, contained within parentheses. I've used `$null` for the parameters I don't want to change and provided my new password as the eighth parameter. The method accepts more parameters, but because I don't want to change the ninth, tenth, or eleventh ones, I can omit them entirely (I could also have specified `$null` for the last three parameters).

This is definitely a complicated syntax. Figure 13.3 breaks it down for you.

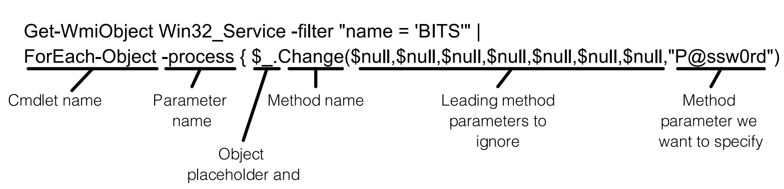



**Figure 13.3 Breaking down the `ForEach-Object` cmdlet**

This exact same pattern can be used for any WMI method. Why would you ever use `Invoke-WmiMethod` instead? Well, it usually does work, and it's a bit easier to type and read. But if you'd prefer to only have to memorize one way of doing things, this `ForEach-Object` way works fine too. I have to caution you, however, that the examples you see on the internet might be a lot less easy to read. PowerShell gurus tend to use aliases, positional parameters, and shortened parameter names a lot, which reduces readability (but saves on typing). Here's my same command again in super-short form:

```
PS C:\> gwmi win32_service -fi "name = 'BITS'" |
➥ % {$_.change($null,$null,$null,$null,$null,$null,$null,"P@ssw0rd") }
```

Here's what I changed:

- I used the alias `Gwmi` instead of `Get-WmiObject`.
- I abbreviated `-filter` to `-fi`.
- I used the `%` alias instead of `ForEach-Object`. Yes, the percent sign is an alias to that cmdlet. I find that to be tough to read, myself, but lots of folks use it.
- I removed the `-process` parameter name again, because it's a positional parameter.

I don't like using aliases and abbreviated parameter names when I'm sharing scripts, posting them in my blog, and so forth, because it makes them too difficult for someone else to read. If you're going to be saving something in a script file, it's worth your time to type everything out (or use Tab completion to let the shell type it out for you).

If you ever wanted to use this example, there are only a few things you might change. Figure 13.4 summarizes the changes you would make:

- You would change the WMI class name, and your filter criteria, to retrieve whatever WMI objects you wanted.
- You would modify the method name from `Change` to whatever method name you wanted to execute.
- You would modify the method's parameter (also called *argument*) list to whatever your method needed. This is always a comma-separated list contained within parentheses. It's okay for the parentheses to be completely empty for methods that have no parameters, such as the `EnableDHCP()` method I introduced earlier in this chapter.

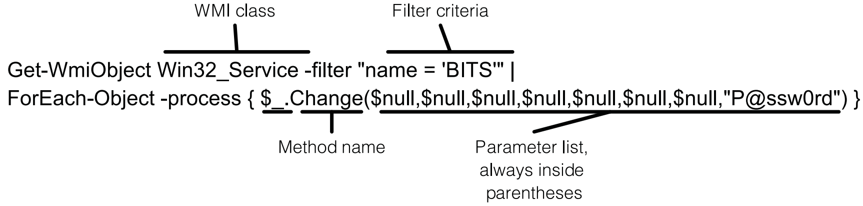



**Figure 13.4 The changes you would make to the example in order to execute a different WMI method**

## 13.5 Common points of confusion

The techniques we've covered in this chapter are among the most difficult ones in PowerShell, and they often cause the most confusion in my classes. Let me try to highlight some of the problems students tend to run into, and provide some alternative explanations that will hopefully help you avoid the same issues.

### 13.5.1 Which way is the right way?

I use the terms *batch cmdlet* or *action cmdlet* to refer to any cmdlet that performs some action against a group, or collection, of objects all at once. In other words, rather than you having to instruct the computer, "Go through this list of things, and perform this one action with each of those things," you just send the whole group to a cmdlet, and the cmdlet handles it.

Microsoft is getting better about providing these kinds of cmdlets with their products, but the coverage isn't 100 percent yet (and probably won't be for many years, because there are so many complex Microsoft products). But when a cmdlet does exist, I prefer to use it. That said, other PowerShell people prefer other ways, depending on what they learned first and what they remember most easily. All of the following are exactly the same:

```
Get-Service -name *B* | Stop-Service                          ❶ Batch cmdlet

Get-Service -name *B* | ForEach-Object { $_.Stop() }          ❷ ForEach-Object

Get-WmiObject Win32_Service -filter "name LIKE '%B%' |        ❸ WMI
➥ Invoke-WmiMethod -name StopService

Get-WmiObject Win32_Service -filter "name LIKE '%B%' |        ❹ WMI and ForEach-Object
➥ ForEach-Object { $_.StopService() }
```

Here's how each works:

- The first approach is to use a batch cmdlet ❶. Here, I'm using `Get-Service` to retrieve all services with a "B" in their name, and then stop them.
- The second approach is similar. Rather than using a batch cmdlet, however, I'm piping the services to `ForEach-Object`, and asking it to execute each service's `Stop()` method ❷.
- The third way is to use WMI, rather than the shell's native service-management cmdlets ❸. I'm retrieving the desired services (again, any with "B" in their name), and piping them to `Invoke-WmiMethod`. I'm telling it to invoke the `StopService` method, which is the method name that the WMI service objects use.
- The fourth way uses `ForEach-Object` instead of `Invoke-WmiMethod`, but accomplishes exactly the same thing ❹. This is a combination of ❷ and ❸, not a whole new way of doing things.

Heck, there's even a fifth way, using PowerShell's scripting language, that does the same thing! There are lots of ways to accomplish almost anything in PowerShell, and none of them are wrong. Some are just easier to learn, remember, and repeat than others, which is why I've focused on the techniques I have, in the order that I did.

There's yet another way, because the `Stop-Service` cmdlet can be directly told which processes to stop:

```
Stop-Process -name BITS
```

I didn't include this in the preceding list, because the `-name` parameter of `Stop-Service` doesn't accept wildcards, so it can't do exactly what the other examples are doing.

Those examples also illustrate some important differences between using native cmdlets and WMI:

- Native cmdlets' filtering criteria usually use `*` as a wildcard character, where WMI filtering uses the percent sign (`%`)—don't confuse that percent sign for the `ForEach-Object` alias! This percent sign is enclosed within the value of `Get-WmiObject`'s `-filter` parameter, and it isn't an alias.
- Native objects often have similar capabilities to WMI ones, but the syntax may differ. Here, the `ServiceController` objects produced by `Get-Service` have a `Stop()` method; when I access those same services through the WMI `Win32_Service` class, the method name becomes `StopService()`.
- Native filtering often uses native comparison operators, such as `-eq`; WMI uses programming-style operators such as `=` or `LIKE`.

Which do you use? It doesn't matter, because there is no one right way. You may even end up using a mix of these, depending on the circumstances and the capabilities that the shell is able to offer you for the task at hand.

### 13.5.2 WMI methods versus cmdlets

When do you use a WMI method or a cmdlet to accomplish a task? It's a simple choice:

- If you retrieved something by using `Get-WmiObject`, you'll take action on that something by using a WMI method. You can execute the method by using `Invoke-WmiMethod` or the `ForEach-Object` approach.
- If you retrieved something by using an approach other than `Get-WmiObject`, you'll use a native cmdlet to take action against that something. Or, if whatever you retrieved has a method but no supporting cmdlet, you might use the `ForEach-Object` approach to execute that method.

Notice that the lowest common denominator here is `ForEach-Object`: its syntax is perhaps the most difficult, but it can always be used to accomplish whatever needs to be done.

You can never pipe anything to a method. You can only pipe from one cmdlet to another. If a cmdlet doesn't exist to do what you need, but a method does, then you pipe to `ForEach-Object`, and have it execute the method.

For example, suppose you retrieve something using a `Get-Something` cmdlet. You want to delete that something, but there's no Delete-Something or Remove-Something cmdlet. The Something objects do, however, have a `Delete` method. You can do this:

```
Get-Something | ForEach-Object { $_.Delete() }
```

### 13.5.3 Method documentation

Always remember that methods are revealed by piping objects to `Get-Member`. Again let's use the fictional `Get-Something` cmdlet as an example:

```
Get-Something | Get-Member
```

WMI methods aren't documented in PowerShell's built-in help system; you'll need to use a search engine (usually searching on the WMI class name) to locate WMI method instructions and examples. Methods of non-WMI objects are also not available in PowerShell's built-in help system. For example, if you get a member list for a service object you can see that methods named `Stop` and `Start` exist:

```
TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
RequiredServices          AliasProperty RequiredServices = ServicesDepe...
Disposed                  Event         System.EventHandler Disposed(Sy...
Close                     Method        System.Void Close()
Continue                  Method        System.Void Continue()
CreateObjRef              Method        System.Runtime.Remoting.ObjRef ...
Dispose                   Method        System.Void Dispose()
Equals                    Method        bool Equals(System.Object obj)
ExecuteCommand            Method        System.Void ExecuteCommand(int ...
GetHashCode               Method        int GetHashCode()
GetLifetimeService        Method        System.Object GetLifetimeService()
GetType                   Method        type GetType()
InitializeLifetimeService Method        System.Object InitializeLifetim...
Pause                     Method        System.Void Pause()
Refresh                   Method        System.Void Refresh()
Start                     Method        System.Void Start(), System.Voi...
Stop                      Method        System.Void Stop()
ToString                  Method        string ToString()
WaitForStatus             Method        System.Void WaitForStatus(Syste...
```

To find the documentation for these, focus on the `TypeName`, which in this case is `System.ServiceProcess.ServiceController`. Search for that complete type name in a search engine, and you'll usually come across the official developer documentation for that type, which will lead to the documentation for whatever specific method you're after.

### 13.5.4 ForEach-Object confusion

The `ForEach-Object` cmdlet has a punctuation-heavy syntax, and adding in a method's own syntax can create a pretty ugly command line. Here are some tips for breaking any mental logjams:

- Try to use the full cmdlet name instead of its `%` or `ForEach` alias. The full name can be easier to read. If you're using someone else's example, replace aliases with the full cmdlet names.
- The script block enclosed in curly braces executes once for each object that's piped into the cmdlet.
- Within the script block, the `$_` represents one of the objects that was piped in.
- Use `$_` by itself to work with the entire object that was piped in; follow `$_` with a period to work with individual methods or properties.
- Method names are always followed by parentheses, even if the method doesn't require any parameters. When parameters are required, they're delimited by commas and included within the parentheses.

## 13.6 Lab

See if you can answer the following questions and complete the specified tasks. This is an especially important lab, because it draws on skills that you've learned in many previous chapters, and that you should be continuing to use and reinforce as you progress through the remainder of this book.

1. What method of a `ServiceController` object (produced by `Get-Service`) will pause the service without stopping it completely?
2. What method of a `Process` object (produced by `Get-Process`) would terminate a given process?
3. What method of a WMI `Win32_Process` object would terminate a given process?
4. Write four different commands that could be used to terminate all processes named "Notepad," assuming that multiple processes might be running under that same name.

14

# Security alert!

By now, you've probably started to get a feel for how powerful PowerShell can be—and started wondering if maybe all of that power might be a security problem. Of course it *might* be! My goal in this chapter is to help you understand exactly how PowerShell can impact security in your environment, and to show you how PowerShell can be configured to provide precisely the balance of security and power that you require.

## 14.1 Keeping the shell secure

When PowerShell was introduced in late 2006, Microsoft didn't exactly have a spotless record on security and scripting. After all, VBScript and Windows Script Host (WSH) were probably two of the most popular virus and malware vectors of the time, serving as entry points for such popular viruses as "I Love You," "Melissa," and many others. When the PowerShell team announced that they were creating a new command-line shell that would offer unprecedented power and functionality, and would offer scripting capabilities, I'm sure alarms went off, people were evacuated from buildings, and everyone gnashed their teeth in dismay.

But it's okay. PowerShell was created after the famous "Trustworthy Computing Initiative" that Bill Gates started within Microsoft. That initiative had a real effect within the company: each product team is required to have a skilled software security expert sit in on their design meetings, code reviews, and so forth. That expert is referred to as—and I'm not making this up—the product's "Security Buddy." PowerShell's Security Buddy was one of the authors of *Writing Secure Code*, Microsoft's own bible for writing software that's less easily exploited by attackers. You can be assured that PowerShell is as secure as any such product can possibly be—at least, it's that secure by default. Obviously, you can change the defaults, but when you do so, you should consider the security ramifications, not just the functional ones. That's what this chapter is going to help you do.

## 14.2 Windows PowerShell security goals

We need to be clear on what PowerShell does and doesn't do when it comes to security, and the best way to do that is to outline some of PowerShell's security goals.

First and foremost, PowerShell doesn't apply any additional layers of permissions on anything it touches. That means PowerShell will only enable you to do what you already have permission to do. If you can't create new users in Active Directory by using the graphical console, you won't be able to do so in PowerShell either. PowerShell is simply another means of exercising whatever permissions you already have.

PowerShell is also not a way of bypassing any existing permissions. Let's say you want to deploy a script to your users, and you want that script to do something that your users don't normally have permission to do. Well, that script isn't going to work for them. If you want your users to do something, you need to give them permission to do so; PowerShell can only accomplish what the person running a command or script is already permitted to accomplish.

### Above and beyond

It's beyond the scope of this book, but I do want you to be aware that there *are* ways to let your users execute a script that runs under credentials other than their own. This is typically accomplished through a technique called *script packaging*, and it's a feature of some commercial script development environments, such as SAPIEN PrimalScript (www.primaltools.com).

After creating a script, you use the packager to bundle the script into an executable (.EXE) file. This isn't compilation in the programming sense of the term: the executable isn't standalone and does require that PowerShell be installed in order to run. You can configure the packager to encrypt alternative credentials into the executable. That way, when someone runs the executable, it launches the packaged script under whatever credentials you specify, rather than the user's own credentials.

The packaged credentials aren't 100 percent safe. The package does include the username and password, although most packagers encrypt them pretty well. It's safe to say that most office users won't be able to discover the username and password, but it's completely possible for a skilled encryption expert to decrypt the username and password.

PowerShell's security system isn't designed to prevent anyone from typing in, and running, whatever commands they have permission to execute. The idea is that it's pretty difficult to trick a user into typing a long, complicated command, so PowerShell doesn't apply any security beyond the user's existing permissions. We know from past experience, however, that it's easy to trick users into running a script, which might well contain commands that are malicious. So most of PowerShell's security is designed with the goal of preventing users from *unintentionally* running scripts. The "unintentionally" part is very important: nothing in PowerShell's security is intended to prevent a determined user from running a script. The idea is only to prevent users from being *tricked* into running scripts from untrusted sources.

PowerShell's security is also not a defense against malware. Once you have malware on your system, that malware can do anything you have permission to do. It might use PowerShell to execute malicious commands, but it might just as easily use any of a dozen other techniques to damage your computer. Once you have malware on your system, you're "owned," and PowerShell isn't a second line of defense. That means you'll continue to need anti-malware software to prevent malware from getting onto your system in the first place. This is a hugely important concept that a lot of people miss: just because a piece of malware might utilize PowerShell to do harm doesn't make that malware PowerShell's problem. The malware must be stopped by your anti-malware software. Nothing in PowerShell is designed or intended to protect an already-compromised system.

## 14.3 Execution policy and code signing

The first security measure PowerShell includes is an *execution policy*. This is a machine-wide setting that governs the scripts that PowerShell will execute. As I stated before, the intent of this setting is to help prevent users from being tricked into running a script. The default setting, in fact, is `Restricted`, which prevents scripts from being executed at all. That's right: by default, PowerShell can be used to interactively run commands, but it can't be used to run scripts. If you try, you'll get this error message:

```
File C:\test.ps1 cannot be loaded because the execution of scripts is disa
bled on this system. Please see "get-help about_signing" for more details.
At line:1 char:7
+ ./test <<<<
    + CategoryInfo          : NotSpecified: (:) [], PSSecurityException
    + FullyQualifiedErrorId : RuntimeException
```

### 14.3.1 Execution policy settings

You can view the current execution policy by running `Get-ExecutionPolicy`. The execution policy can be changed in one of three ways:

- By running the `Set-ExecutionPolicy` command. This changes the setting in the `HKEY_LOCAL_MACHINE` portion of the Windows registry, and so must usually be run by an Administrator, because normal users don't have permission to write to that portion of the registry.
- By using a Group Policy object. Windows Server 2008 R2 comes with the Windows PowerShell–related settings built right in; for older domain controllers you

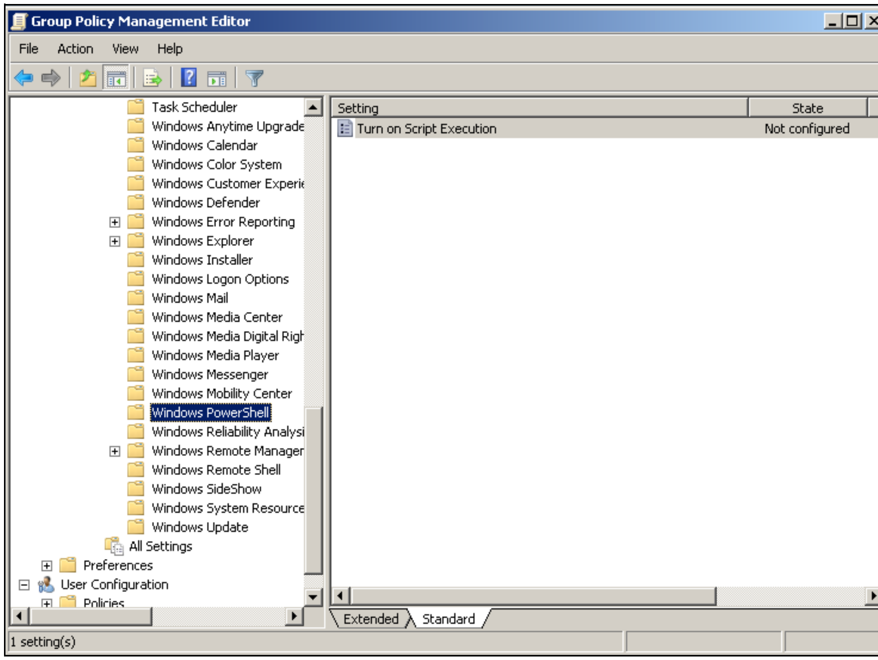



**Figure 14.1 Finding the Windows PowerShell settings in a Group Policy object**

can download an ADM template to extend Group Policy. You'll find it at http://mng.bz/U6tJ. You can also just visit http://download.microsoft.com and punch in "PowerShell ADM" as a search term.

The PowerShell settings are located under Computer Configuration\Policies\Administrative Templates\Windows Components\Windows PowerShell as shown in Figure 14.1. Figure 14.2 shows the policy setting enabled. When configured via a Group Policy object, the setting in the Group Policy will override any local setting. In fact, if you try to run `Set-ExecutionPolicy`, it will work, but a warning message will tell you that your new setting had no effect due to a Group Policy override.

- By manually running PowerShell.exe and using its `-ExecutionPolicy` command-line switch. When run in this fashion, the specified execution policy will override any local setting as well as any Group Policy–defined setting.

The execution policy can be set to one of five settings (note that the Group Policy object only provides access to the middle three):

- `Restricted`—This is the default, and scripts aren't executed. The only exceptions are a few Microsoft-supplied scripts that set up PowerShell's default configuration settings. Those scripts carry a Microsoft digital signature and won't execute if modified.

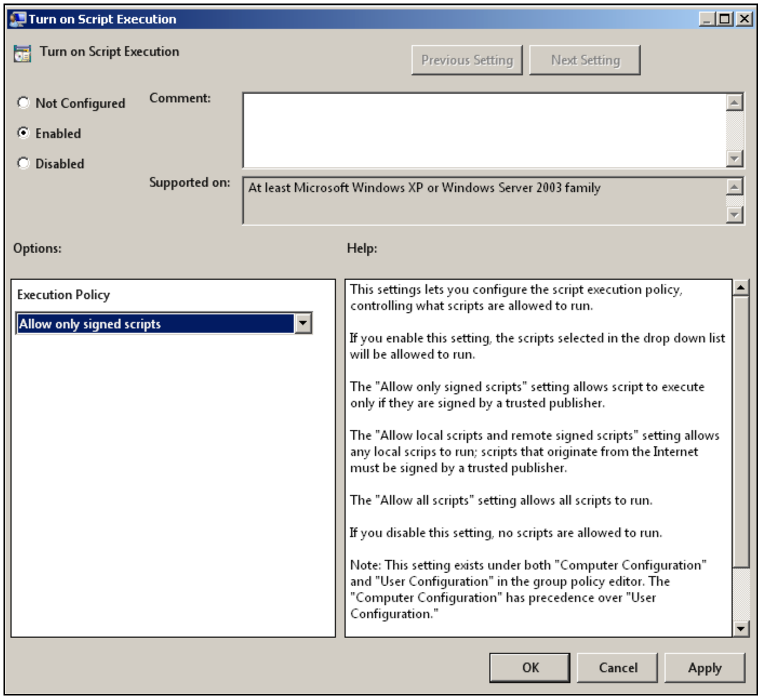



**Figure 14.2 Changing the Windows PowerShell execution policy in a Group Policy object**

- `AllSigned`—PowerShell will execute any script that has been digitally signed by using a code-signing certificate that was issued by a trusted Certification Authority (CA).
- `RemoteSigned`—PowerShell will execute any local script, and will execute remote scripts if they have been digitally signed by using a code-signing certificate that was issued by a trusted CA. "Remote scripts" are those that exist on a remote computer, usually accessed by a Universal Naming Convention (UNC) path. Scripts marked as having come from the internet are also considered "remote"; Internet Explorer, Firefox, and Outlook all mark downloads as having come from the internet. Some versions of Windows can distinguish between internet paths and UNC paths; in those cases, UNC paths on the local network aren't considered "remote."
- `Unrestricted`—All scripts will run. I don't like or recommend this setting, because it provides too little protection.
- `Bypass`—This is a special setting that's intended to be used by application developers who are embedding PowerShell within their application. This setting bypasses the configured execution policy and should be used only when the hosting application is providing its own layer of script security.

Microsoft recommends that `RemoteSigned` be used when you want to run scripts, and that it be used only on computers where scripts must be executed. All other computers should be left at `Restricted`. `RemoteSigned` is felt to provide a good balance between security and convenience; `AllSigned` is stricter but does require that all of your scripts be digitally signed. Which means we should probably discuss what digital signing is all about.

### 14.3.2 Digital code signing

Digital code signing, *code signing* for short, is the process of applying a cryptographic signature to a text file. Signatures appear at the end of the file and look something like this:

```
<!-- SIG # Begin signature block -->
<!-- MIIXXAYJKoZIhvcNAQcCoIIXTTCCF0kCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB -->
<!-- gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR -->
<!-- AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUJ7qroHx47PI1dIt4lBg6Y5Jo -->
<!-- UVigghIxMIIEYDCCA0ygAwIBAgIKLqsR3FD/XJ3LwDAJBgUrDgMCHQUAMHAxKzAp -->
<!-- YjcCn4FqI4n2XGOPsFq7OddgjFWEGjP1O5igggyiX4uzLLehpcur2iC2vzAZhSAU -->
<!-- DSq8UvRB4F4w45IoaYfBcOLzp6vOgEJydg4wggR6MIIDYqADAgECAgphBieBAAAA -->
<!-- ZngnZui2t++Fuc3uqv0SpAtZIikvz0DZVgQbdrVtZG1KVNvd8d6/n4PHgN9/TAI3 -->
<!-- an/xvmG4PNGSdjy8Dcbb5otiSjgByprAttPPf2EKUQrFPzREgZabAatwMKJbeRS4 -->
<!-- kd6Qy+RwkCn1UWIeaChbs0LJhix0jm38/pLCCOo1nL79E1sxJumCe6GtqjdWOIBn -->
<!-- KKe66D/GX7eGrfCVg2Vzgp4gG7fHADFEh3OcIvoILWc= -->
<!-- SIG # End signature block -->
```

The signature contains two important pieces of information: First, it lists the identity of the company or organization that signed the script. Second, it includes an encrypted copy of the script, which PowerShell can decrypt. Understanding how this works requires a bit of background information, which will also help you make some important decisions about security in your environment.

In order to create a digital signature, you need to have a code-signing certificate. Also referred to as Class 3 certificates, these are available from commercial CAs like GoDaddy, VeriSign, Thawte, CyberTrust, and others. You might also obtain one from your company's internal Public Key Infrastructure (PKI), if you have one. Class 3 certificates are normally issued only to organizations and companies, not to individuals, although your company may issue them internally to specific users. Before issuing a certificate, the CA is responsible for verifying the identity of the recipient—the certificate is essentially a kind of digital identification card, listing the holder's name and other details. So before issuing a certificate to XYZ Corporation, a CA needs to verify that an authorized representative of XYZ Corporation is making the request. This verification process is the single most important step in the entire security framework, and you should only trust a CA that you know does a good job of verifying the identities of the companies to which it issues certificates. If you're not familiar with a CA's verification procedures, you *should not trust* that CA.

Trust is configured in Windows' Internet Options control panel (and can also be configured by Group Policy). In that control panel, select the Content tab, and then click

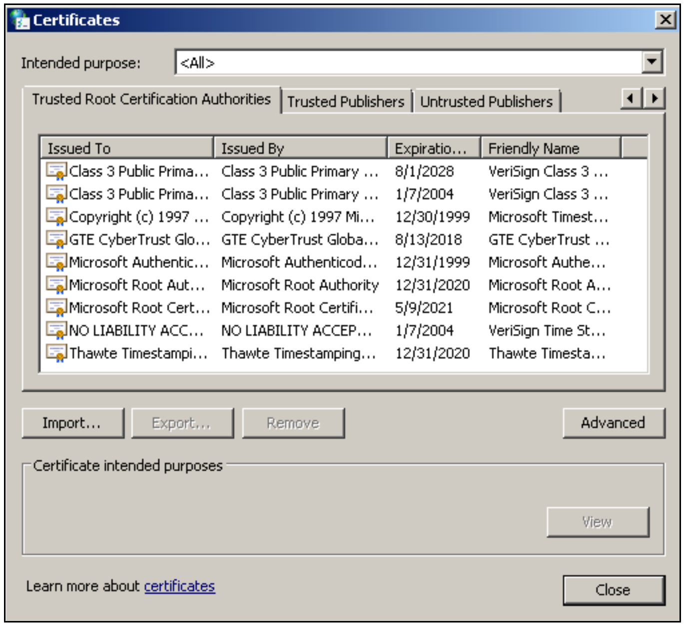



**Figure 14.3 Configuring your computer's Trusted Root Certification**

Publishers. In the resulting dialog box, select the Trusted Root Certification Authorities tab. As shown in figure 14.3, you'll see a list of the CAs that your computer trusts.

When you trust a CA, you also trust all certificates issued by it. If someone uses a certificate to sign a malicious script, you can use the signature itself to track down the author—that's why signed scripts are considered more "trusted" than unsigned scripts. But if you place your trust in a CA that does a bad job of verifying identities, a malicious script author might be able to obtain a fraudulent certificate, and you wouldn't be able to use their signature to track them down. That's why choosing which CAs to trust is such a big responsibility.

Once you have obtained a Class 3 certificate (specifically, you need one packaged as an Authenticode certificate—CAs usually offer different varieties for different operating systems and programming languages), you install it on your computer. Once installed, you can then use PowerShell's `Set-AuthenticodeSignature` cmdlet to apply a digital signature to a script. Run `help about_signing` in the shell to learn more about how to do that. Many commercial script development environments (PrimalScript, PowerShell Plus, PowerGUI, and others) can also apply signatures, and can even do so automatically when you save a script, making the signing process more transparent for you.

Signatures not only provide information about the script author's identity; they also ensure that the script hasn't been modified since the author signed it. It works like this:

1. The script author holds a digital certificate, which consists of two cryptographic keys: a public key and a private key.

2 When signing a script, the signature is encrypted using the private key. Only the script author has access to that key, and only the public key can decrypt the signature. The signature contains a copy of the script.
3 When PowerShell runs the script, it uses the author's public key (which is included along with the signature) to decrypt the signature. If the decryption fails, the signature was tampered with, and the script won't run. If the copy of the script within the signature doesn't match the clear-text copy, the signature is considered broken, and the script won't run.

Figure 14.4 illustrates the entire process that PowerShell goes through when trying to run a script. You can see how the AllSigned execution policy is thus somewhat more secure: under that setting, only scripts containing a signature will execute, meaning that you'll always be able to identify a script's author. Of course, you'll also have to sign every script you want to run, and re-sign them any time you change them, which can be inconvenient.

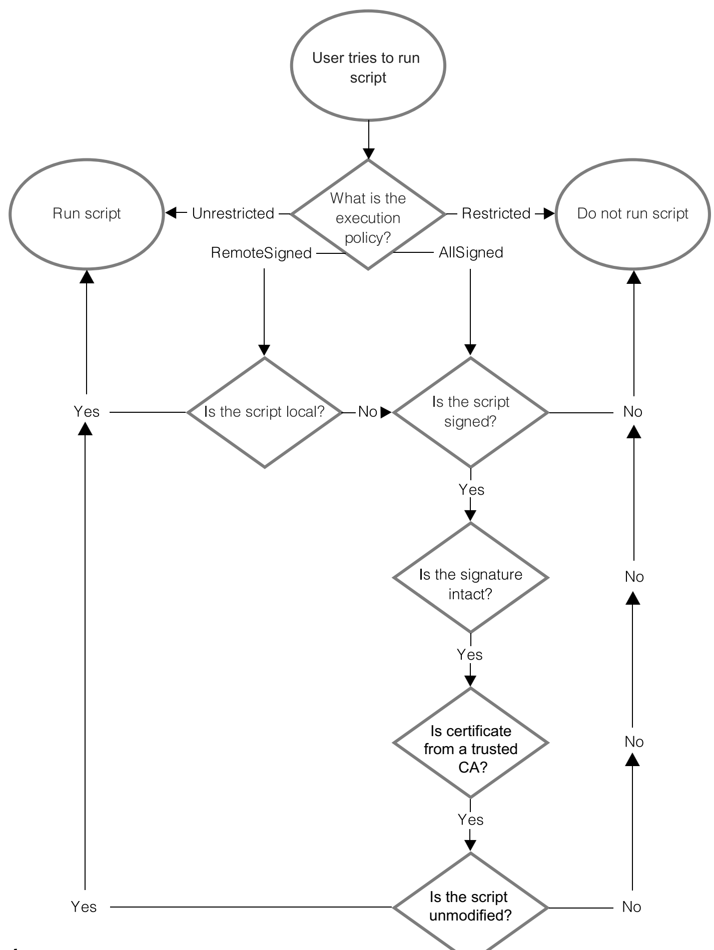



**Figure 14.4 The process PowerShell follows when attempting to execute a script**

## 14.4 Other security measures

PowerShell has two other key security measures that are in effect at all times, and that should not be modified.

First, the .PS1 filename extension (which is what the shell uses to identify PowerShell scripts) isn't considered an executable file type by Windows. Double-clicking a .PS1 file will normally open it in Notepad for editing, rather than attempting to execute it. This configuration is intended to help prevent users from unknowingly executing a script, even if the execution policy would allow it.

Second, you can't run a script within the shell by simply typing its name. The shell never searches the current directory for scripts, so if you have a script named test.ps1, simply changing to its folder and typing `test` or `test.ps1` won't run the script.

Here's an example:

```
PS C:\> test
The term 'test' is not recognized as the name of a cmdlet, function, scrip
t 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.
At line:1 char:5
+ test <<<<
    + CategoryInfo          : ObjectNotFound: (test:String) [], CommandNo
   tFoundException
    + FullyQualifiedErrorId : CommandNotFoundException

Suggestion [3,General]: The command test was not found, but does exist in t
he current location. Windows PowerShell doesn't load commands from the curr
ent location by default. If you trust this command, instead type ".\test".
See "get-help about_Command_Precedence" for more details.
PS C:\>
```

As you can see, PowerShell does detect the script but warns you that you have to type either an absolute or relative path in order to run the script. Because the script is located in C:\, you could run either `C:\test`, which is an absolute path, or run `.\test`, which is a relative path that points to the current folder.

The purpose of this security feature is to guard against a type of attack called *command hijacking*. The attack involves putting a script into a folder, and giving it the same name as a built-in command, such as `Dir`. With PowerShell, you never put a path in front of a command name. So if you run `Dir`, you know you're running the command. If you run `.\Dir`, you know you're running a script named Dir.ps1.

## 14.5 Other security holes?

As I've already written, PowerShell's security is primarily focused on preventing users from unknowingly running an untrusted script. There's nothing to stop a user from manually typing commands into the shell, or even from copying the entire contents of a script and pasting them into the shell (although the commands might not have the exact same effect when run in that fashion). It's a little more difficult to convince a user to do that, and to explain to them how to do it, so Microsoft didn't focus on that scenario as a potential attack vector. Just remember that PowerShell doesn't grant your users any additional permissions: they will only be able to do those things that you've permitted them to do.

Sure, someone could call a user on the phone, or send them an email, and walk them through the process of opening PowerShell, typing a few commands, and damaging their computer. But that same someone could also call a user and walk them through the attack using means other than PowerShell. It would be just as easy (or difficult, depending on your viewpoint) to convince a user to open Explorer, select their Program Files folder, and hit Delete on their keyboard. Actually, that would be easier than walking them through the equivalent PowerShell command. I point this out only because people tend to get nervous about the command line and its seemingly infinite reach and power, but the fact is that you and your users can't do anything with the shell that couldn't be done in a half-dozen other ways.

## 14.6 Security recommendations

As I mentioned earlier, Microsoft recommends the use of the `RemoteSigned` execution policy for computers where you need to run scripts. I disagree and suggest that you consider using `AllSigned`. Yes, it's a bit less convenient, but you can make it more convenient by following these two recommendations:

- Commercial CAs charge up to $900 per year for a code-signing certificate. If you don't have an internal PKI that can provide a free one, you can make your own. Run `help about_signing` for information on obtaining and using Makecert.exe, a tool that will make a certificate that will be trusted only by your local computer. If that's the only place where you need to run scripts, it's a quick and free way to obtain a certificate.
- Edit scripts in one of the editors I mentioned, each of which can sign the script for you each time you save the file. That makes the signing process transparent and automatic, making it more convenient.

As I've already stated, I don't think you should change the .PS1 filename association. I've seen some folks modify Windows to recognize .PS1 as an executable, meaning that you can double-click a script to run it. That takes us right back to the bad old days of VBScript, and you probably want to avoid doing that.

I want to point out that none of the scripts I supply on MoreLunches.com are digitally signed. That means it's possible for those to be modified without my (or your) knowledge, so before you run any of those scripts, you need to take the time to review them, understand what they're supposed to be doing, and make sure they match what's in this book (if appropriate). I didn't sign the scripts specifically because I *want you to take that time*: you should be in the habit of carefully reviewing anything you download from the internet, no matter how "trusted" the author may seem.

## 14.7 Lab

Your task in this lab is simple—so simple, in fact, that I won't even post a sample solution on MoreLunches.com. I just want you to configure your shell to allow script execution. Use the `Set-ExecutionPolicy` cmdlet, and I suggest using the `RemoteSigned` policy setting. You're welcome to use `AllSigned`, but it will be impractical for the purposes of this book's remaining labs. You could also choose `Unrestricted`, but I don't ever recommend the use of that setting because it's too liberal.

That said, if you're using PowerShell in a production environment, please make sure that whatever execution policy setting you choose is compatible with your organization's security rules and procedures. I don't want you getting in trouble for the sake of this book and its labs!

# 15 Variables: a place to store your stuff

I've already mentioned that PowerShell contains a scripting language, and in a few more chapters we're going to start playing with it. Once you start scripting, however, you tend to start needing *variables*, so we'll get those out of the way in this chapter. Variables can be used in a lot of places other than long, complex scripts, so I'll also show you some practical ways in which you can utilize them.

## 15.1 Introduction to variables

A simple way to think of a variable is as a box in the computer's memory that has a name. You can put whatever you want into the box: a single computer name, a collection of services, an XML document, or whatever you like. You access the box by using its name, and when accessing it you can either put things in it, add things to it, or retrieve things from it (when you do so, those things actually stay in the box, so that you can retrieve them over and over).

PowerShell doesn't require a lot of formality around variables. For example, you don't have to explicitly announce or declare your intention to use a variable before doing so. The types of the contents of the variable can be changed: one moment, you might have a single process in it, and the next moment you can store a bunch of computer names in it. A variable can even contain multiple different things, such as a collection of services *and* a collection of processes (although I admit that using the variable's contents, in those cases, can be tricky).

## 15.2 Storing values in variables

Everything in PowerShell—and I do mean *everything*—is treated as an object. Even a simple string of characters, such as a computer name, is considered an object. For example, piping a string to `Get-Member` (or its alias, `Gm`) reveals that the object is of the type `System.String` and that it has a great many methods that you can work with (I'm truncating the list here to save space):

```
PS C:\> "SERVER-R2" | gm

   TypeName: System.String

Name              MemberType          Definition
----              ----------          ----------
Clone             Method              System.Object Clone()
CompareTo         Method              int CompareTo(System.Object valu...
Contains          Method              bool Contains(string value)
CopyTo            Method              System.Void CopyTo(int sourceInd...
EndsWith          Method              bool EndsWith(string value), boo...
Equals            Method              bool Equals(System.Object obj), ...
GetEnumerator     Method              System.CharEnumerator GetEnumera...
GetHashCode       Method              int GetHashCode()
GetType           Method              type GetType()
GetTypeCode       Method              System.TypeCode GetTypeCode()
IndexOf           Method              int IndexOf(char value), int Ind...
IndexOfAny        Method              int IndexOfAny(char[] anyOf), in...
```

> **TRY IT NOW** Try running this same command in PowerShell so that you can see the complete list of methods—and even a property—that comes with a `System.String`.

Although that string is technically an object, just like everything else in the shell, you'll find that folks tend to refer to it as a simple *value*. That's because, in most cases, what you're concerned about is the string itself—`"SERVER-R2"` in my example—and you're less concerned about retrieving information from properties. That's different from, say, a process, where the entire process object is this big, abstract data construct, and what you're usually dealing with are individual properties such as `VM`, `PM`, `Name`, `CPU`, `ID`, and so forth. I guess you could say that a `String` is an object, but it's a much less complicated object than something like a `Process`.

PowerShell allows you to store these simple values in a variable. To do so, specify the variable, and use the equal sign operator—the *assignment* operator—followed by whatever you want to put within the variable. Here's an example:

```
PS C:\> $var = "SERVER-R2"
```

> **TRY IT NOW** You'll want to follow along with these examples, so that you can replicate the results I'll demonstrate. You should use your test server's name rather than SERVER-R2.

It's important to note that the dollar sign (`$`) isn't part of the variable's name. In this example, the variable name is `var`. The dollar sign is a cue to the shell that what follows is going to be a variable name, and that we want to access the contents of that variable. In this case, we're setting the contents of the variable.

Here are some points to keep in mind about variables and their names:

- Variable names usually contain letters, numbers, and underscores, and it's most common for them to begin with a letter or underscore.
- Variable names can contain spaces, but the name must be enclosed in curly braces. For example, `${My Variable}` is the way to represent a variable named "My Variable." Personally, I really dislike variable names that contain spaces, because they require more typing and they're harder to read.
- Variables don't persist between shell sessions. When you close the shell, any variables you created will be gone.
- Variable names can be quite long—long enough that you don't need to worry about how long. Try and make variable names sensible. For example, if you'll be putting a computer name into a variable, use `computername` as the variable name. If a variable will contain a bunch of processes, then `processes` is a good variable name.
- Except for folks who have a VBScript background, PowerShell users don't typically utilize variable name prefixes to indicate what is stored in the variable. For example, in VBScript, `strComputerName` was a common kind of variable, indicating that the variable stored a string (the "str" part). PowerShell doesn't care if you do that, but it's no longer considered a desirable practice by the community at large.

To retrieve the contents of a variable, use the dollar sign followed by the variable name. Again, the dollar sign tells the shell that you want to access the *contents* of a variable; following it with the variable name tells the shell which variable you're accessing.

```
PS C:\> $var
SERVER-R2
```

A variable can be used in place of a value in almost any situation. For example, when using WMI, you have the option to specify a computer name. The command might normally look like this:

```
PS C:\> get-wmiobject win32_computersystem -comp SERVER-R2

Domain              : company.pri
Manufacturer        : VMware, Inc.
Model               : VMware Virtual Platform
Name                : SERVER-R2
PrimaryOwnerName    : Windows User
TotalPhysicalMemory : 3220758528
```

You can substitute a variable for the same thing:

```
PS C:\> get-wmiobject win32_computersystem -comp $var

Domain              : company.pri
Manufacturer        : VMware, Inc.
```

```
Model                : VMware Virtual Platform
Name                 : SERVER-R2
PrimaryOwnerName     : Windows User
TotalPhysicalMemory  : 3220758528
```

By the way, I realize that `var` is a pretty generic variable name. I'd normally use `computername`, but in this specific instance I plan to reuse `$var` in several situations, so I decided to keep it generic. Don't let this example stop you from using more sensible variable names in real life.

I may have put a string into `$var` to begin with, but I can change that anytime I want:

```
PS C:\> $var = 5
PS C:\> $var | gm

   TypeName: System.Int32

Name          MemberType   Definition
----          ----------   ----------
CompareTo     Method       int CompareTo(System.Object value), int CompareT...
Equals        Method       bool Equals(System.Object obj), bool Equals(int ...
GetHashCode   Method       int GetHashCode()
GetType       Method       type GetType()
GetTypeCode   Method       System.TypeCode GetTypeCode()
ToString      Method       string ToString(), string ToString(string format...
```

Here, I placed an integer into `$var`, and then I piped `$var` to `Gm`. You can see that the shell now recognizes the contents of `$var` as a `System.Int32`, or a 32-bit integer.

## 15.3 Fun tricks with quotes

Now that we're talking about variables, it's a good time to cover a neat PowerShell feature. So far in this book, I've advised that you generally enclose strings within single quotation marks. There's a reason for that: PowerShell treats everything enclosed in single quotation marks as a literal string.

Consider this example:

```
PS C:\> $var = 'What does $var contain?'
PS C:\> $var
What does $var contain?
```

Here, you can see that the `$var` within single quotes is treated as a literal. In double quotation marks, however, that's not the case.

Check out this trick:

```
PS C:\> $computername = 'SERVER-R2'
PS C:\> $phrase = "The computer name is $computername"
PS C:\> $phrase
The computer name is SERVER-R2
```

I started by storing `SERVER-R2` in the variable `$computername`. Next, I stored `"The computer name is $computername"` in the variable `$phrase`. When I did so, I used double quotes. PowerShell will automatically seek out dollar signs within double quotes, and replace any variables it finds *with their contents*. So when I displayed the contents of `$phrase`, `$computername` was replaced with `SERVER-R2`, the contents of the variable.

This replacement action only happens when the string is initially parsed by the shell. So, right now, `$phrase` contains `"The computer name is SERVER-R2"`—it doesn't contain the `"$computername"` string. I can test that by trying to change the contents of `$computername`, and seeing if `$phrase` updates itself:

```
PS C:\> $computername = 'SERVER1'
PS C:\> $phrase
The computer name is SERVER-R2
```

The `$phrase` variable stayed the same.

Another facet of this double quotes trick is the PowerShell escape character. This character is the backtick (`` ` ``), and on a U.S. keyboard it's located on one of the upper-left keys, usually below the Escape key and usually on the same key as the tilde (`~`) character. The problem is that, in some fonts, it's practically indistinguishable from a single quote. In fact, I usually configure my shell to use the Consolas font, because it makes it a little bit easier to distinguish the backtick than the Lucida Console or Raster font.

> **TRY IT NOW** Click the control box in the upper-left corner of your PowerShell window, and select Properties. On the Font tab, select the Consolas font. Click OK, and then type a single quote and a backtick so that you can see the difference between these characters. Figure 15.1 shows what it looks like on my system. Can't see the difference? I barely can, either, even when using a pretty large font size. It's a tough distinction, but make sure you're comfortable distinguishing between them in whatever font face and size you select.

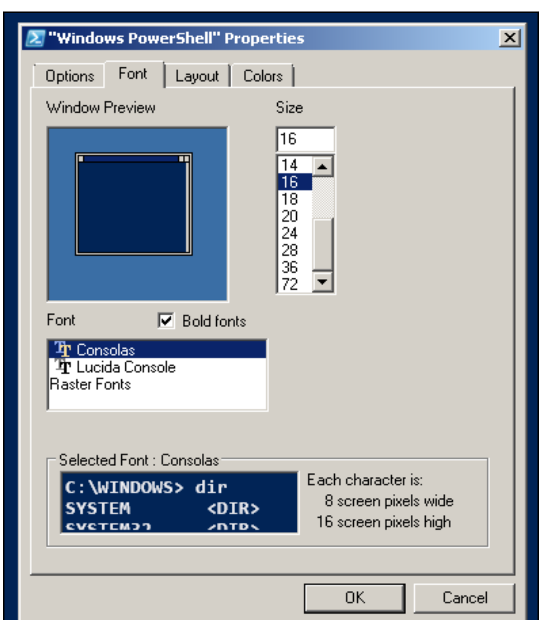



**Figure 15.1 Setting a font that makes it easier to distinguish the backtick character from the single quote**

So what does this escape character do? It removes whatever special meaning might be associated with the character after it, or in some cases, it adds special meaning to the following character. Here's an example of this first usage:

```
PS C:\> $computername = 'SERVER-R2'
PS C:\> $phrase = "`$computername contains $computername"
PS C:\> $phrase
$computername contains SERVER-R2
```

When I assigned the string to `$phrase`, I used `$computername` twice. The first time, I preceded the dollar sign with a backtick. Doing so took away the dollar sign's special meaning as a variable indicator, and made it a literal dollar sign. You can see in the final output, on the last line, that `$computername` was stored in the variable. I didn't use the backtick the second time, so `$computername` was replaced with the contents of that variable.

Now, here's an example of the second way a backtick can work:

```
PS C:\> $phrase = "`$computername`ncontains`n$computername"
PS C:\> $phrase
$computername
contains
SERVER-R2
```

Look carefully, and you'll notice two `` `n `` in the phrase—one after the first `$computername` and one after `contains`. In this example, the backtick is adding a special meaning. Normally, an "n" is just a letter, but with the backtick in front of it, it becomes a carriage return and line feed (think *n* for *new line*).

Run `help about_escape` for more information, including a list of other special escape characters. You can, for example, use an escaped "t" to insert a tab, or an escaped "a" to make your computer beep (think *a* for *alert*).

## 15.4 Storing lots of objects in a variable

At this point, we've been working with variables that contain a single object, and those objects have all been simple values. We've worked directly with the object itself, rather than with properties or methods. Now, let's try putting a bunch of objects into a variable.

One way to do so is to use a comma-separated list, because PowerShell recognizes those lists as collections of objects:

```
PS C:\> $computers = 'SERVER-R2','SERVER1','localhost'
PS C:\> $computers
SERVER-R2
SERVER1
Localhost
```

Notice that I was careful to put the commas outside the quotation marks. If I'd put them inside, I would have had a single object that included commas and three computer names. This way, I get three distinct objects, all of which are `String` types. As you can see, when I examined the contents of the variable, PowerShell displayed each object on its own line.

You can also access individual elements, one at a time. To do so, specify an index number for the object you want, in square brackets. The first object is always at index number 0, the second is at index number 1, and so forth. You can also use an index of `-1` to access the last object, `-2` for the next-to-the-last object, and so on.

```
PS C:\> $computers[0]
SERVER-R2
PS C:\> $computers[1]
SERVER1
PS C:\> $computers[-1]
localhost
PS C:\> $computers[-2]
SERVER1
```

The variable itself has a property that lets you see how many objects are in it:

```
PS C:\> $computers.count
3
```

Beyond that special property, you can access the properties and methods of the objects inside the variable as if they were properties and methods of the variable itself. This is a bit easier to see, at first, with a variable that contains a single object:

```
PS C:\> $computername.length
9
PS C:\> $computername.toupper()
SERVER-R2
PS C:\> $computername.tolower()
server-r2
PS C:\> $computername.replace('R2','2008')
SERVER-2008
PS C:\> $computername
SERVER-R2
```

Here, I'm using the `$computername` variable that I created earlier in the chapter. If you remember, that variable contained an object of the type `System.String`, and you should have seen the complete list of properties and methods of that type when you piped a string to `Gm`. I've used the `Length` property, as well as the `ToUpper()`, `ToLower()`, and `Replace()` methods. In each case, I have to follow the method name with parentheses, even though neither `ToUpper()` nor `ToLower()` require any parameters inside those parentheses. Also, none of these methods changed what was in the variable—you can see that on the last line. Instead, each method created a new `String` based on the original one, as modified by the method.

When a variable contains multiple objects, this gets a bit trickier. Even if every object inside the variable is of the same type, as is the case with my `$computers` variable, *you can't call a method, or access a property, on multiple objects at the same time.* If you try to do so, you'll get an error:

```
PS C:\> $computers.toupper()
Method invocation failed because [System.Object[]] doesn't contain a metho
d named 'toupper'.
```

```
At line:1 char:19
+ $computers.toupper <<<< ()
    + CategoryInfo          : InvalidOperation: (toupper:String) [], Runt
   imeException
    + FullyQualifiedErrorId : MethodNotFound
```

Instead, you have to specify which object within the variable you want, and then access a property or execute a method:

```
PS C:\> $computers[0].tolower()
server-r2
PS C:\> $computers[1].replace('SERVER','CLIENT')
CLIENT1
```

Again, these methods are producing new strings, not changing the ones inside the variable. You can test that by examining the contents of the variable:

```
PS C:\> $computers
SERVER-R2
SERVER1
Localhost
```

What if you wanted to change the contents of the variable? You'd simply assign a new value into one of the existing objects:

```
PS C:\> $computers[1] = $computers[1].replace('SERVER','CLIENT')
PS C:\> $computers
SERVER-R2
CLIENT1
Localhost
```

You can see that I changed the second object in the variable, rather than producing a new string.

By the way, I want to show you two other options for working with the properties and methods of a bunch of objects contained in a variable. The previous couple of examples only executed methods on a single object within the variable. If I wanted to run the `ToLower()` method on every object within the variable, and store the results back into the variable, I would do this:

```
PS C:\> $computers = $computers | ForEach-Object { $_.ToLower() }
PS C:\> $computers
server-r2
client1
localhost
```

This is a bit complicated, so let's break it down in figure 15.2. I started the pipeline with `$computers =`, which means the results of the pipeline will be stored in that variable. Those results will overwrite whatever was in the variable previously. The pipeline begins with `$computers` being piped to `ForEach-Object`. The cmdlet will enumerate each object in the pipeline (I have three computer names, which are `String` objects), and execute its script block for each. Within the script block, the `$_` placeholder will contain one piped-in object at a time, and I'm executing the `ToLower()` method of

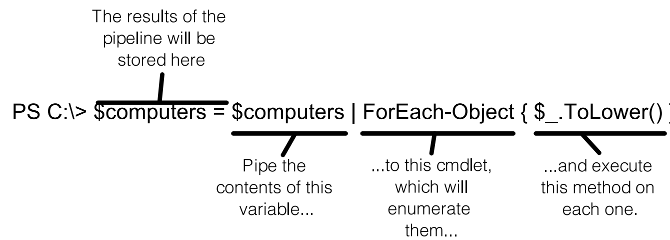



**Figure 15.2 Using `ForEach-Object` to execute a method against each object contained within a variable**

each object. The new `String` objects produced by `ToLower()` will be placed into the pipeline—and into the `$computers` variable.

You can do something similar with properties, by using `Select-Object`. This will select the `Length` property of each object that I pipe to the cmdlet:

```
PS C:\> $computers | select-object length

                                                                Length
                                                                ------
                                                                     9
                                                                     7
                                                                     9
```

Because the property is numeric, PowerShell right-aligns the output.

## 15.5 Declaring a variable's type

So far, we've just stuck objects into variables and let PowerShell figure out what kind of object was what. The fact is that PowerShell doesn't care what kinds of objects get put into the box. You, however, might care.

For example, suppose you have a variable that you expect to contain a number. You plan to do some arithmetic with that number, and you ask a user to input that number. Here's an example, which you can type directly into the command line:

```
PS C:\> $number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number = $number * 10
PS C:\> $number
100100100100100100100100100100
```

> **TRY IT NOW** I haven't showed you `Read-Host` yet—I'm saving it for the next chapter—but its operation should be pretty obvious if you follow along with this example.

What the heck? 100 multiplied by 10 is 100100100100100100100100100100? What crazy New Math is that?

If you're sharp-eyed, you may have spotted what's happening. PowerShell didn't treat my input as a number; it treated it as a string. Instead of multiplying 100 by 10, PowerShell *duplicated the string 100 ten times*. So the result is the string `100`, listed ten times in a row. Oops.

We can verify that the shell is in fact treating the input as a string:

```
PS C:\> $number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number | gm

   TypeName: System.String

Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone()
CompareTo        Method               int CompareTo(System.Object valu...
Contains         Method               bool Contains(string value)
```

Yep, piping `$number` to `Gm` confirms that the shell sees it as a `System.String`, not a `System.Int32`. There are a couple of ways that we could choose to deal with this problem, and the easiest for me is the one we'll use right now. I'm going to tell the shell that the `$number` variable should contain an integer, which will force the shell to try to convert any input to an actual number. I do that by specifying the desired data type, `int`, in square brackets immediately prior to the variable's first use:

```
PS C:\> [int]$number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number | gm

   TypeName: System.Int32

Name        MemberType  Definition
----        ----------  ----------
CompareTo   Method      int CompareTo(System.Object value), int CompareT...
Equals      Method      bool Equals(System.Object obj), bool Equals(int ...
GetHashCode Method      int GetHashCode()
GetType     Method      type GetType()
GetTypeCode Method      System.TypeCode GetTypeCode()
ToString    Method      string ToString(), string ToString(string format...

PS C:\> $number = $number * 10
PS C:\> $number
1000
```

**1 Force variable to [int]**

**2 Confirm that variable is Int32**

**3 Variable was treated as number**

Here, I've used `[int]` to force `$number` to contain only integers 1. After entering my input, I pipe `$number` to `Gm` to confirm that it is indeed an integer, not a string 2. At the end, I can see that the variable was treated as a number and real multiplication took place 3.

Another benefit of this technique is that the shell will throw an error if it can't convert the input into a number, because `$number` is only capable of storing integers:

```
PS C:\> [int]$number = Read-Host "Enter a number"
Enter a number: Hello
Cannot convert value "Hello" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:13
+ [int]$number <<<<  = Read-Host "Enter a number"
```

```
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformati
   onMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

That's a great way to help prevent problems later on down the line, because you're assured that `$number` will contain the exact type of data you expect it to.

There are many different object types that you can use in place of `[int]`, but these are some of the ones you'll use most commonly include:

- `[int]`—Integer numbers
- `[single]` and `[double]`—Single-precision and double-precision floating numbers (numbers with a decimal portion)
- `[string]`—A string of characters
- `[char]`—Exactly one character (as in, `[char]$c = 'X'`)
- `[xml]`—An XML document; whatever string you assign to this will be parsed to make sure it contains valid XML markup (for example, `[xml]$doc = Get-Content MyXML.xml`)
- `[adsi]`—An Active Directory Services Interface (ADSI) query; the shell will execute the query and place the resulting object or objects into the variable (such as `[adsi]$user = "WinNT:\\MYDOMAIN\Administrator,user"`)

Specifying an object type for a variable is a great way to prevent certain tricky logic errors in more complex scripts. Once you specify the object type, PowerShell enforces it until you explicitly retype the variable:

```
PS C:\> [int]$x = 5
PS C:\> $x = 'Hello'
Cannot convert value "Hello" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:3
+ $x <<<<  = 'Hello'
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformati
   onMetadataException
    + FullyQualifiedErrorId : RuntimeException

PS C:\> [string]$x = 'Hello'
PS C:\> $x | gm

   TypeName: System.String

Name                MemberType           Definition
----                ----------           ----------
Clone               Method               System.Object Clone()
CompareTo           Method               int CompareTo(System.Object valu...
```

**1** Declares $x as integer

**2** Creates error by putting string into $x

**3** Retypes $x as string

**4** Confirms new type of $x

Here, you can see that I started by declaring `$x` as an integer **1**, and placing an integer into it. When I tried to put a string into it **2**, PowerShell threw an error because it couldn't convert that particular string into a number. Later I retyped `$x` as a string, and was able to put a string into it **3**. I confirmed that by piping the variable to `Gm` and checking its type name **4**.

## 15.6 Commands for working with variables

So far, we've just started using variables without formally declaring our intention to do so. PowerShell doesn't require advanced variable declaration, and you can't force it to do so (VBScript folks who are looking for something like `Option Explicit` will be disappointed; PowerShell has something called `Set-StrictMode`, but it isn't exactly the same thing). The shell does, however, include commands for working with variables:

- `New-Variable`
- `Set-Variable`
- `Remove-Variable`
- `Get-Variable`
- `Clear-Variable`

The thing is, you don't need to use any of these except perhaps `Remove-Variable`, which is good for permanently deleting a variable (you can also use the `Del` command within the VARIABLE: drive to delete a variable). Every other function—creating new variables, reading variables, setting variables—can be done using the ad hoc syntax that I've used so far in this chapter, and there are no specific advantages to using these cmdlets in most cases.

If you do decide to use these cmdlets, you'll give your variable name to the cmdlets' `-name` parameter. This is *only the variable name*—not including the dollar sign! The one time you might want to use one of these cmdlets is if you're working with something called an *out-of-scope* variable. Messing with out-of-scope variables is a really poor practice, and I'm not going to cover it here, but you'll see it come up in chapter 17.

## 15.7 Variable best practices

I've mentioned most of these practices already, but this is a good time to quickly review them:

- Keep variable names meaningful, but succinct. `$computername` is a great variable name, because it's clear and concise. `$c` is a poor name, because it isn't clear what it contains. `$computer_to_query_for_data` is a bit long for my tastes. Sure, it's meaningful, but do you really want to type that over and over?
- Don't use spaces in variable names. I know you can, but it's ugly syntax.
- If a variable will only contain one kind of object, then declare that when you first use the variable. Doing so can help prevent some confusing logic errors, and if you're working in a commercial script development environment (PrimalScript is the example I'm thinking of), the editor software can provide code-hinting features when you tell it what type of object a variable will contain.

## 15.8 Common points of confusion

The biggest single point of confusion I see new students struggle with is the variable name. Hopefully, I've done a good job of explaining it in this chapter, but always remember that the dollar sign *isn't part of the variable's name*. It's a cue to the shell that you want to access the *contents* of a variable; what follows the dollar sign is taken as the variable's name.

The shell has two parsing rules that let it capture the variable name:

- If the character immediately after the dollar sign is a letter, number, or underscore, the variable name consists of all the characters following the dollar sign, up to the next white space (which might be a space, tab, or carriage return).
- If the character immediately after the dollar sign is an opening curly brace, `{`, the variable name is everything after that curly brace up to, but not including, the closing curly brace, `}`.

## 15.9 Lab

Flip back to chapter 12 and refresh your memory on working with background jobs. For this lab, I'd like you to create a background job that queries the Win32_BIOS information from two computers (use your computer's name and "localhost" if you only have one computer to experiment with). When the job finishes running, I want you to receive the results of the job into a variable. Then, display the contents of that variable. Finally, export the variable's contents to a CliXML file.

## 15.10 Ideas for on your own

Take a few moments and skim through some of the previous chapters in this book. Given that variables are primarily designed to store something that you might use more than once, can you locate anything in the previous chapters where you might find a use for variables?

For example, in chapter 10 you learned to create connections to remote computers. What you did in that chapter was create, use, and close a connection more or less in one step; wouldn't it be useful to create the connection, store it in a variable, and use it for several commands? That's just one instance of where variables can come in handy (and I'm going to show you how to do that in chapter 18), so see if you can find any more examples.

# 16 Input and output

So far in this book, we've primarily been relying on PowerShell's native ability to output tables and lists. As you start to combine commands into more complex scripts, you'll probably want to gain more precise control over what's displayed. You may also have a need to prompt a user for input. In this chapter, you'll learn how to collect that input, and how to display whatever output you might desire.

## 16.1 Prompting for, and displaying, information

How PowerShell displays and prompts for information depends on how PowerShell is being run. You see, PowerShell is built as a kind of under-the-hood engine.

What you interact with is called a *host application.* The command-line console that you see when running PowerShell.exe is often called the *console host.* The graphical PowerShell ISE is usually called the *ISE host* or the *graphical host.* Other non-Microsoft applications can host the shell's engine, as well. You interact with the hosting application, and it passes your commands through to the engine. Whatever results the engine produces are displayed by the hosting application.

Figure 16.1 illustrates the relationship between the engine and the various hosting applications. Each hosting application is responsible for physically displaying any output produced by the engine, and for physically collecting any input requested by the engine. That means output may be displayed, and input collected, in different ways. In fact, the console host and ISE use very different methods for collecting input: the console host presents a text prompt within the command line, but the ISE produces a pop-up dialog box with a text entry area and an OK button.

I wanted to point out these differences because it can sometimes seem confusing to newcomers. Why would one command behave one way in the command-line

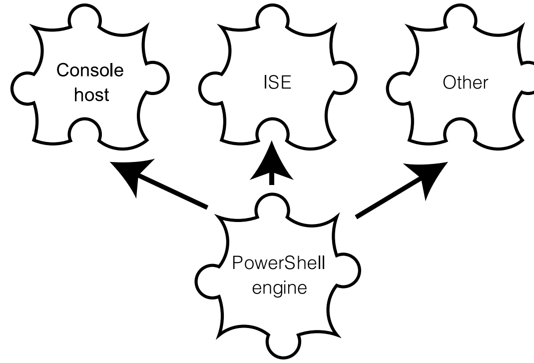



**Figure 16.1 Various applications are capable of hosting the PowerShell engine**

window but behave completely differently in the ISE? It's because the way in which you interact with the shell is determined by the hosting application, and not by PowerShell itself.

## 16.2 Read-Host

PowerShell's `Read-Host` cmdlet is designed to display a text prompt and then collect text input from the user. You saw me use this for the first time in the previous chapter, so the syntax may seem familiar:

```
PS C:\> read-host "Enter a computer name"
Enter a computer name: SERVER-R2
SERVER-R2
```

This example highlights two important facts about the cmdlet:

- A colon is added to the end of the prompt.
- Whatever the user types is returned as the result of the command (technically, it's placed into the pipeline).

You'll often capture the input into a variable, which looks like this:

```
PS C:\> $computername = read-host "Enter a computer name"
Enter a computer name: SERVER-R2
```

> **TRY IT NOW** Time to start following along. At this point, you should have a valid computer name in the `$computername` variable. Don't use SERVER-R2 unless that's the name of the computer you're working on!

As I wrote earlier, the ISE will display a dialog box, rather than prompting directly within the command line, as shown in figure 16.2. Other hosting applications, including script editors like PowerGUI, PowerShell Plus, or PrimalScript, will each have their own way of implementing `Read-Host`.

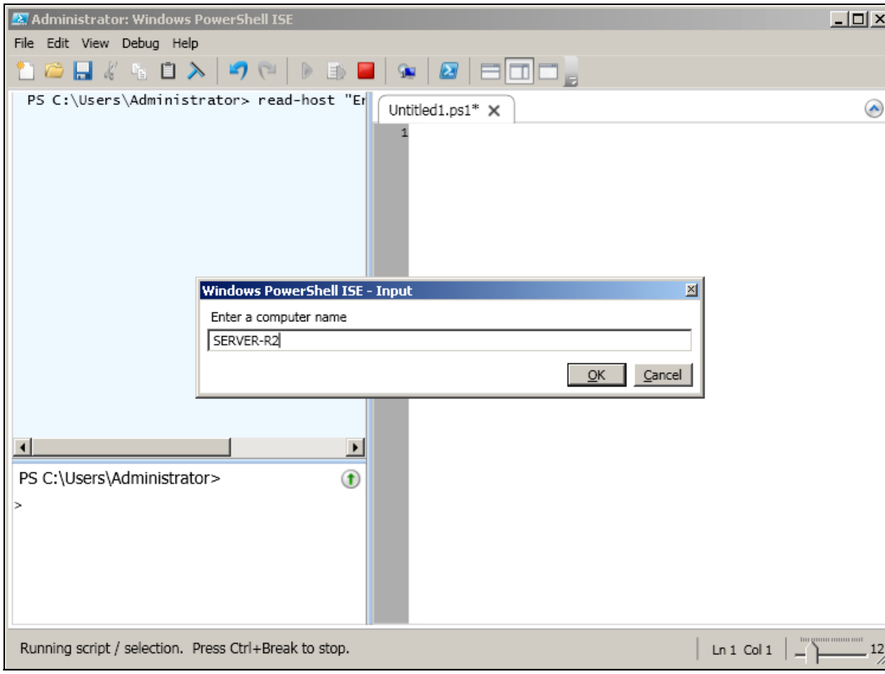



**Figure 16.2 The ISE displays a dialog box for `Read-Host`.**

There isn't much else to say about `Read-Host`: it's a useful cmdlet, but not a particularly exciting one. In fact, after introducing `Read-Host` in most classes, the usual question is, "Is there a way to always display a graphical input box?" Many administrators want to deploy scripts to their users, and they don't want them to have to enter information into a command-line interface (it isn't very "Windows-like," after all). The answer is, yes, but it isn't very straightforward. The final result is shown in figure 16.3.

To do this, we're going to have to dive into the .NET Framework itself. You'll start with this command:

```
PS C:\> [void][System.Reflection.Assembly]::LoadWithPartialName('Microsoft
➥ .VisualBasic')
```

Type that all as a single command. You only have to do this once in a given shell session, but it doesn't hurt to run the command a second time.

This command loads a portion of the .NET Framework, Microsoft.VisualBasic, that PowerShell doesn't automatically load. This portion of the framework contains most of the Visual Basic–centric framework elements, including things like graphical input boxes. Here's what the command is doing:

- The `[void]` part is converting the result of the command into the `void` data type. You learned how to do this kind of conversion with integers in the previous chapter; the `void` data type is a special one that means "throw the result

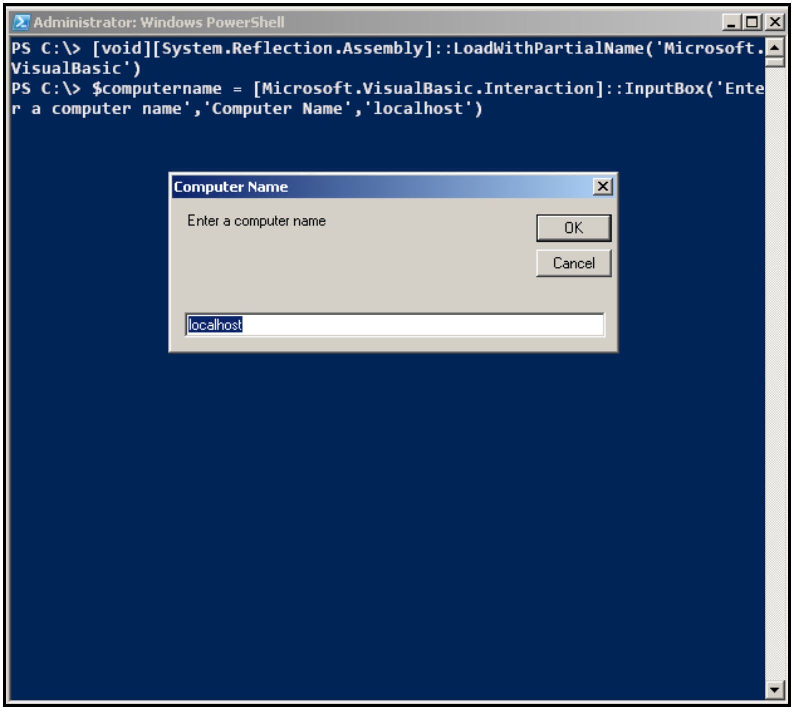



**Figure 16.3 Creating a graphical input box in Windows PowerShell**

away." In other words, we don't want to see the result of this command, so we convert the result to `void`. Another way to do the same thing would be to pipe the result to `Out-Null`.

- Next, we're accessing the `System.Reflection.Assembly` type, which represents our application (which is PowerShell). I've enclosed the type name in square brackets, as if I were declaring a variable to be of that type. But rather than declaring a variable, we're using two colons to access a *static method* of the type. Static methods exist without us having to create an instance of the type.
- The static method we're using is `LoadWithPartialName()`, which accepts the name of the framework component I want to load.

If all of that is as clear as mud, don't worry; you can use the command as-is without needing to understand how it works. Once the right bits of the framework are loaded, you can use them, and that's done like this:

```
PS C:\> $computername = [Microsoft.VisualBasic.Interaction]::InputBox('Ente
➥ r a computer name','Computer Name','localhost')
```

I'm using a static method again, from the `Microsoft.VisualBasic.Interaction` type, which I just loaded into memory with the previous command. Again, if the "static method" stuff doesn't make sense, don't worry—use this command as-is.

The three bits you can change are the parameters of the `InputBox()` method:

- The first parameter is the text for your prompt.
- The second parameter is the title for the prompt's dialog box.
- The third parameter, which can be left blank or omitted entirely, is the default value that you want prefilled in the input box.

This is definitely more complicated than using `Read-Host`, but if you insist on a dialog box, this is the best way to achieve that.

## 16.3 Write-Host

Now that you can collect input, you'll want some way of displaying output. The `Write-Host` cmdlet is one way—not always the best way, but it's available to you, and it's important that you understand how it works.

As figure 16.4 illustrates, `Write-Host` runs in the pipeline like any other cmdlet, but it doesn't place anything into the pipeline. Instead, it writes directly to the hosting application's screen. Because it does that, it's able to use alternate foreground and background colors, through its `-foregroundColor` and `-backgroundColor` command-line parameters.

> **TRY IT NOW** You'll definitely want to run this command yourself to see the colorful results.

```
PS C:\> write-host "COLORFUL!" -fore yellow -back magenta
COLORFUL!
```

`Write-Host` should usually be used only when you need to display a specific message, perhaps using color to draw attention to it. This isn't the appropriate way to produce normal output from a script or command.

For example, you should never use `Write-Host` to manually format a table—there are better ways to produce the output, using techniques that enable PowerShell itself

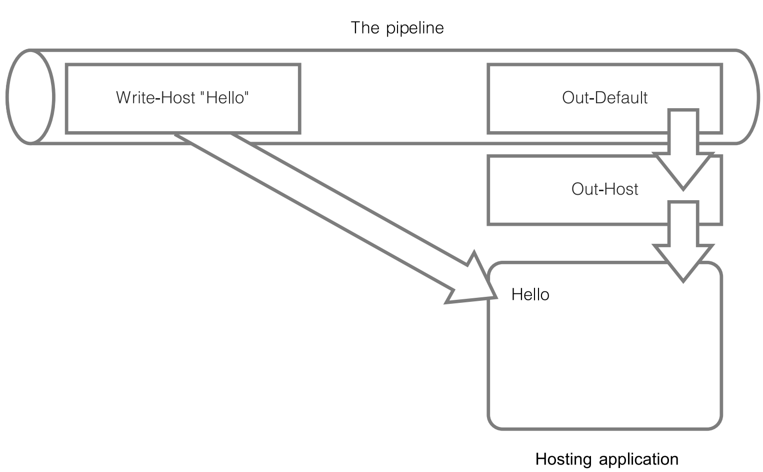



**Figure 16.4 `Write-Host` bypasses the pipeline and writes directly to the hosting application's display.**

to handle the formatting. We won't be covering those techniques in this chapter, but in chapter 19 you'll play with them extensively. `Write-Host` is also not the best way to produce error messages, warnings, debugging messages, and so on—again, there are more specific ways to do those things, and you'll see those in this chapter. You probably won't use `Write-Host` much, if you're using the shell correctly.

## 16.4 Write-Output

Unlike `Write-Host`, `Write-Output` sends objects into the pipeline. Because it isn't writing directly to the display, it doesn't permit you to specify alternative colors or anything. In fact, `Write-Output` (or its alias, `Write`) isn't technically designed to display output at all. As I said, it sends objects into the pipeline—it's the pipeline itself that eventually displays those objects. Figure 16.5 illustrates how this works.

Refer back to chapter 8 for a quick review of how objects go from the pipeline to the screen. Here's the basic process:

1. `Write-Output` puts the `String` object `"Hello"` into the pipeline.
2. There's nothing else in the pipeline, so `"Hello"` travels to the end of the pipeline, where `Out-Default` always sits.
3. `Out-Default` passes the object to `Out-Host`.
4. `Out-Host` asks PowerShell's formatting system to format the object. Because in this example it's a simple `String`, the formatting system returns the text of the string.
5. `Out-Host` places the formatted result onto the screen.

The results are similar to what you'd get using `Write-Host`, but the object took a very different path to get there. That path is important, because the pipeline could contain other things. For example, consider this command (which you're welcome to try):

```
PS C:\> write-output "Hello" | where-object { $_.length -gt 10 }
```

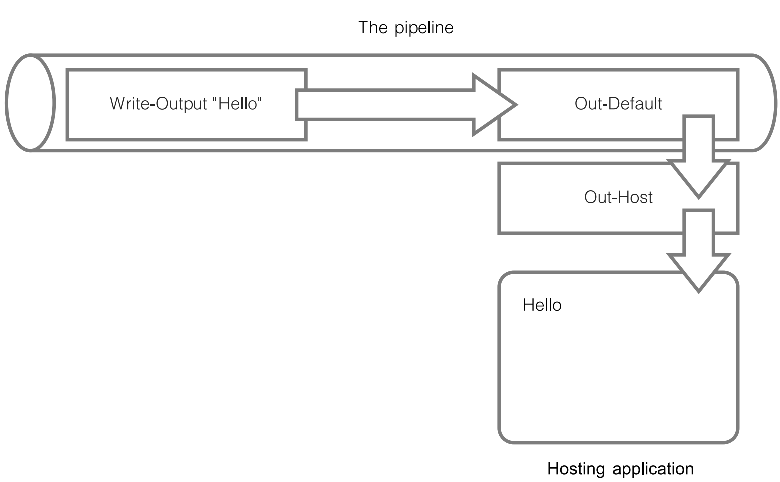



**Figure 16.5 `Write-Output` puts objects into the pipeline, which in some cases will eventually result in those objects being displayed.**

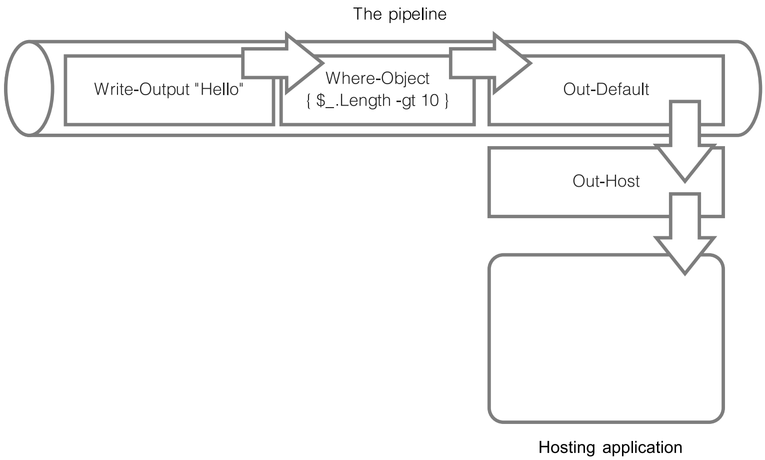



**Figure 16.6 Placing objects into the pipeline means they can be filtered out before they're displayed.**

There's no output from this command, and figure 16.6 illustrates why. `"Hello"` was placed into the pipeline. Before it got to `Out-Default`, however, it had to pass through `Where-Object`, which filtered out anything having a `Length` property of less than or equal to `10`, which in this case included our poor `"Hello"`. So our `"Hello"` got dropped out of the pipeline. There was nothing left in the pipeline for `Out-Default`, so there was nothing to pass to `Out-Host`, so nothing was displayed.

Contrast that command with this one:

```
PS C:\> write-host "Hello" | where-object { $_.length -gt 10 }
Hello
```

All I did was replace `Write-Output` with `Write-Host`. This time, `"Hello"` went directly to the screen, not into the pipeline. `Where-Object` had no input, and produced no output, so nothing was displayed by `Out-Default` and `Out-Host`. But because `"Hello"` had been written directly to the screen, we saw it anyway.

`Write-Output` may seem new, but it turns out you've been using it all along. It's the shell's default cmdlet. When you tell the shell to do something that isn't actually a command, the shell passes whatever you typed to `Write-Output` behind the scenes.

## 16.5 Other ways to write

PowerShell has a few other ways of producing output. None of these write to the pipeline like `Write-Output` does; they work a bit more like `Write-Host`. All of them, however, produce output in a way that can be suppressed.

The shell comes with built-in configuration variables for each of these alternative output methods. When the configuration variable is set to `Continue`, the commands I'm about to show you do indeed produce output. When the configuration variable is set to `SilentlyContinue`, the associated output command produces nothing. Table 16.1 contains the list of commands.

**Table 16.1 Alternative output cmdlets**

| Cmdlet | Purpose | Configuration variable |
|---|---|---|
| `Write-Warning` | Displays warning text, in yellow by default and preceded by the label "WARNING:" | `$WarningPreference` (`Continue` by default) |
| `Write-Verbose` | Displays additional informative text, in yellow by default and preceded by the label "VERBOSE:" | `$VerbosePreference` (`SilentlyContinue` by default) |
| `Write-Debug` | Displays debugging text, in yellow by default and preceded by the label "DEBUG:" | `$DebugPreference` (`SilentlyContinue` by default) |
| `Write-Error` | Produces an error message | `$ErrorActionPreference` (`Continue` by default) |

`Write-Error` works a bit differently because it actually writes an error to PowerShell's error stream. Chapter 22 will discuss errors in a bit more detail, and will provide more information on how `$ErrorActionPreference` can be used.

There's also a `Write-Progress` cmdlet that can display progress bars, but it works entirely differently. Feel free to read its help for more information and for examples, but we won't be covering it in this chapter.

To use any of these cmdlets, first make sure that its associated configuration variable is set to `Continue`. If it's set to `SilentlyContinue`, which is the default for a couple of them, you won't see any output at all. Then, use the cmdlet to output a message.

Note that some PowerShell hosting applications may display the output from these cmdlets in a different location. In PrimalScript, for example, debugging text is written to a different output pane than the script's main output, so that the debug text can be more easily separated for analysis. You'll see more about `Write-Debug` in chapter 23.

## 16.6 Lab

`Write-Host` and `Write-Output` can be a bit tricky to work with. See how many of these tasks you can complete, and if you get completely stuck, it's okay to peek at the sample answers available on MoreLunches.com.

1. Use `Write-Output` to display the result of 100 multiplied by 10.
2. Use `Write-Host` to display the result of 100 multiplied by 10.
3. Prompt the user to enter a name, and then display that name in yellow text.

4 Prompt the user to enter a name, and then display that name only if it's longer than 5 characters. Do this all in a single line—don't use a variable.

That's all for this lab. Because these cmdlets are all pretty straightforward, I want you to spend some more time experimenting with them on your own. Be sure to do that—there are some ideas in the next section.

## 16.7 Ideas for on your own

Spend some time getting comfortable with all of the cmdlets in this chapter. Make sure you can display verbose output, accept input, and even display a graphical input box. You'll be using the commands from this chapter a lot from here on out, so you should read their help files and even jot down a quick syntax reminder for future reference.

# 17 You call this scripting?

So far, you could have accomplished everything in this book using PowerShell's command-line interface. You haven't had to write a single script. That's a big deal for me, because I see a lot of administrators initially shy away from scripting, (rightly) perceiving it as a kind of programming, and (correctly) feeling that learning it can sometimes take more time than it's worth. Hopefully, you've seen how much you can accomplish in PowerShell without having to become a programmer.

But at this point, you may also be starting to feel that constantly retyping the same commands, over and over, is going to become pretty tedious. You're right, so in this chapter we're going to dive into PowerShell scripting—but we're still not going to be programming. Instead, we're going to focus on scripts as little more than a way of saving our fingers from unnecessary retyping.

## 17.1 Not programming ... more like batch files

Most Windows administrators have, at one point or another, created a command-line batch file (which usually have a .BAT or .CMD filename extension). These are nothing more than simple text files that you can edit with Windows Notepad, containing a list of commands to be executed in a specific order. Technically, you call those commands a *script* because, like a Hollywood script, they tell the performer (your computer) exactly what to do and say, and in what order to do and say it. But batch files rarely look like programming, in part because the Cmd.exe shell has a very limited language that doesn't permit incredibly complicated scripts.

PowerShell scripts—or *batch files*, if you prefer—work similarly. Simply list the commands that you want to run, and the shell will execute those commands in the order specified. You can create a script by simply copying a command from the host window and pasting it into Notepad. Of course, Notepad is a pretty horrible text editor. I expect you'll be happier with the PowerShell ISE, or with a third-party editor like PowerGUI, PrimalScript, or PowerShell Plus.

The ISE, in fact, makes "scripting" practically indistinguishable from using the shell interactively. By using the ISE's script pane, you simply type the command or commands you want to run, and then click the Run button in the toolbar to execute those commands. Click Save and you've created a script without having to copy and paste anything at all.

## 17.2 Making commands repeatable

The idea behind PowerShell scripts is, first and foremost, to make it easier to run a given command over and over, without having to manually retype it every time. That being the case, we should come up with a command that we want to run over and over again, and use that as an example throughout this chapter. I want to make this decently complex, so I'll start with something from WMI and add in some filtering, sorting, and other stuff.

At this point, I'm going to switch to using the PowerShell ISE instead of the normal console window, because the ISE will make it easier for me to migrate my command into a script. Frankly, the ISE makes it easier to type complex commands, because you get a full-screen editor instead of working on a single line within the console host.

Here's my command:

```
Get-WmiObject -class Win32_LogicalDisk -computername localhost
➥ -filter "drivetype=3" |
 Sort-Object -property DeviceID |
 Format-Table -property DeviceID,
                  @{l='FreeSpace(MB)';e={$_.FreeSpace / 1MB -as [int]}},
                  @{l='Size(GB';e={$_.Size / 1GB -as [int]}},
                  @{l='%Free';e={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Figure 17.1 shows how I've entered this into the ISE. Notice that I selected the three-pane layout by using the toolbar button set on the far right. Also notice that I formatted my command so that each physical line ends in either a pipe character or a comma. By doing so, I'm forcing the shell to recognize these multiple lines as a single, one-line command. You could do the same thing in the console host, but this formatting is especially effective in the ISE because it makes my command a lot easier to read. Also notice that I've used full cmdlet names and parameter names and that I've specified every parameter name rather than using positional parameters. All of that will make my script easier to read and follow either for someone else, or in the future when I might have forgotten what my original intent was.

I've run the command by clicking the green Run toolbar icon in the ISE (you could also press F5) to test it, and my output shows that it's working perfectly. Here's a neat trick in the ISE: you can highlight a portion of your command and press F8 to just run the highlighted portion. Because I've formatted my command so that there's one distinct command per physical line, that makes it easy for me to test my command bit by bit. I can highlight and run the first line independently. If I'm satisfied with the output,

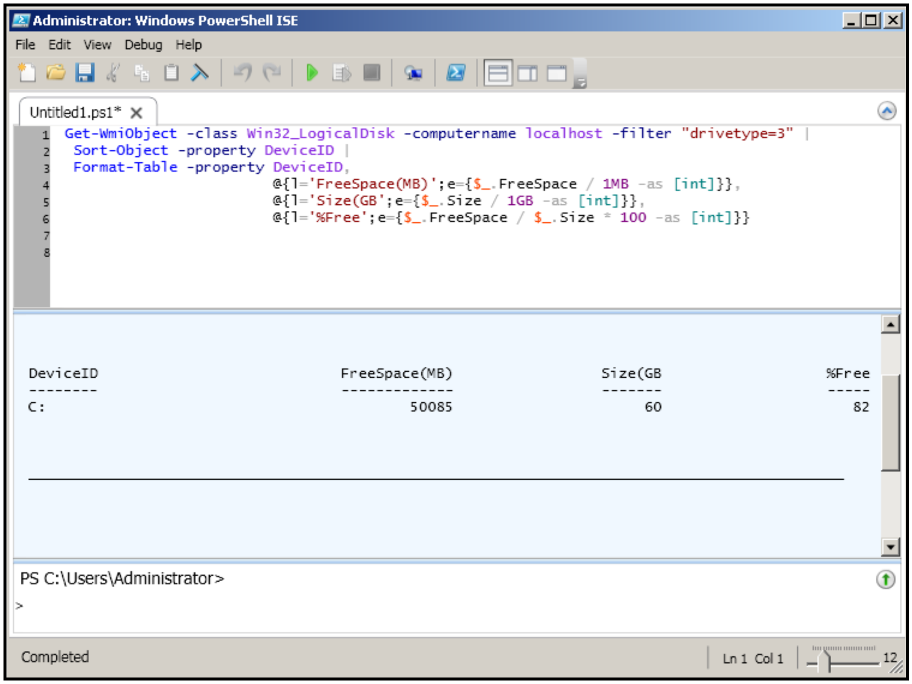



**Figure 17.1 Entering and running a command in the ISE**

I can highlight the first and second lines, and run them. If it worked as expected, I can run the whole command.

At this point, I can save the command—I guess we can start calling it a script now! I'll save it as Get-DiskInventory.ps1. I like giving my scripts cmdlet-style verb-noun names. You can see how this script is going to start to look and work a lot like a cmdlet, so it makes sense to give it a cmdlet-style name.

> **TRY IT NOW** I'm assuming that you have already completed chapter 14 and enabled scripting by setting a more permissive execution policy. If you haven't done so, then you should flip back to chapter 14 and complete its hands-on lab so that scripts will run in your copy of PowerShell.

## 17.3 Parameterizing commands

When you think about running a command over and over, you might realize that some portion of the command is going to have to change from time to time. For example, suppose you wanted to give Get-DiskInventory.ps1 to some of your colleagues, who might be less experienced in using PowerShell. It's a pretty complex, hard-to-type command, and they might appreciate having it bundled into an easier-to-run script. But, as written, the script only runs against the local computer. You can certainly imagine that some of your colleagues might want to get a disk inventory from one or more remote computers instead.

One option would be to have them open up the script and change the `-computername` parameter's value. But it's entirely possible that they wouldn't be comfortable doing so, and there's a chance that they'll change something else and break the script entirely. It would be better to provide a formal way for them to pass in a different computer name (or set of names). At this stage, we need to identify the things that might need to change when the command is run, and replace those things with variables.

We'll set the computer name variable to a static value for now, so that we can still test the script. This listing shows my revised script.

**Listing 17.1 Get-DiskInventory.ps1 with a parameterized command**

```
$computername = 'localhost'
Get-WmiObject -class Win32_LogicalDisk `
 -computername  $computername `
 -filter "drivetype=3" |
 Sort-Object -property DeviceID |
 Format-Table -property DeviceID,
     @{l='FreeSpace(MB)';e={$_.FreeSpace / 1MB -as [int]}},
     @{l='Size(GB';e={$_.Size / 1GB -as [int]}},
     @{l='%Free';e={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

1 Sets new variable
2 Breaks line with backtick
3 Uses variable

I've done three things here, two of which are functional and one of which is purely cosmetic:

- I've added a variable, `$computername`, and set it equal to `localhost` 1. I've noticed that most PowerShell commands that accept a computer name use the parameter name `-computername`, and I want to duplicate that convention, which is why I chose the variable name that I did.
- I've replaced the value for the `-computername` parameter with my variable 3. Right now, the script should run exactly the same as it did before (and I tested to make sure it does), because I've put `localhost` into the `$computername` variable.
- I added a backtick after the `-computername` parameter and its value 2. This escapes, or takes away the special meaning of, the carriage return at the end of the line. That tells PowerShell that the next physical line is part of this same command. You don't need to do that when the line ends in a pipe character or a comma, but in order to fit the code within this book, I needed to break the line a bit before the pipe character. This will only work if the backtick character is the last thing on the line!

Once again, I've been careful to run my script and verify that it's still working. I always do that after making any kind of change, to make sure I haven't introduced some random typo or other error.

## 17.4 Creating a parameterized script

Now that I've identified the elements of my script that might change from time to time, I need to provide a way for someone else to specify new values for those elements. In other words, I need to take that hardcoded `$computername` variable and turn it into an input parameter.

PowerShell makes this really easy, and the next listing shows the result.

**Listing 17.2 Get-DiskInventory.ps1, with an input parameter**

```
param (
  $computername = 'localhost'                        <-- 1 Param block
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
 -filter "drivetype=3" |
 Sort-Object -property DeviceID |
 Format-Table -property DeviceID,
     @{l='FreeSpace(MB)';e={$_.FreeSpace / 1MB -as [int]}},
     @{l='Size(GB';e={$_.Size / 1GB -as [int]}},
     @{l='%Free';e={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

All I did was add a `Param()` block around my variable declaration 1. This defines `$computername` as a parameter, and specifies that `localhost` is the default value to be used if the script is run without a computer name being specified. You don't have to provide a default value, but I like to do so when there's a reasonable value that I can think of.

All parameters declared in this fashion are both named and positional, meaning that I can now run the script from the command line in any of these ways:

```
PS C:\> .\Get-DiskInventory.ps1 server-r2
PS C:\> .\Get-DiskInventory.ps1 -computername server-r2
PS C:\> .\Get-DiskInventory.ps1 -comp server-r2
```

In the first instance, I used the parameter positionally, providing a value but not the parameter name. In the second and third instance, I specified the parameter name, but in the third instance I abbreviated that name in keeping with PowerShell's normal rules for parameter name abbreviation. Note that in all three cases I had to specify a path (`.\`, which is the current folder) to the script, because the shell won't automatically search the current directory to find the script.

You can specify as many parameters as you need to, by separating them with commas. For example, suppose that I wanted to also parameterize the filter criteria. Right now, it's only retrieving logical disks of type 3, which represents fixed disks. I could change that to a parameter, as shown next.

**Listing 17.3 Get-DiskInventory.ps1, with an additional parameter**

```
param (
  $computername = 'localhost',
  $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
 -filter "drivetype=$drivetype" |                <-- Using parameter
 Sort-Object -property DeviceID |
 Format-Table -property DeviceID,
     @{l='FreeSpace(MB)';e={$_.FreeSpace / 1MB -as [int]}},
```

```
    @{l='Size(GB';e={$_.Size / 1GB -as [int]}},
    @{l='%Free';e={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Notice that I took advantage of PowerShell's ability to replace variables with their values inside of double quotation marks (you learned about that trick in chapter 15).

I can run this script in any of the three original ways, although I could also omit either parameter if I wanted to use the default value for it. Here are some permutations:

```
PS C:\> .\Get-DiskInventory.ps1 server-r2 3
PS C:\> .\Get-DiskInventory.ps1 -comp server-r2 -drive 3
PS C:\> .\Get-DiskInventory.ps1 server-r2
PS C:\> .\Get-DiskInventory.ps1 -drive 3
```

In the first instance, I specified both parameters positionally, in the order in which they're declared within the `Param()` block. In the second case, I specified abbreviated parameter names for both. The third time, I omitted `-drivetype` entirely, using the default value of `3`. In the last instance, I left off `-computername`, using the default value of `localhost`.

## 17.5 Documenting your script

Only a truly mean person would create a useful script and not tell anyone how to use it. Fortunately, PowerShell makes it easy to add help into your script, using comments. You're welcome to add typical programming-style comments to your scripts, but if you're using full cmdlet and parameter names, sometimes your scripts' operation will be obvious. By using a special comment syntax, however, you can provide help that mimics PowerShell's own help files.

The next listing shows what I've added to my script.

**Listing 17.4 Adding help to Get-DiskInventory.ps1**

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
param (
  $computername = 'localhost',
  $drivetype = 3
)
```

```
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
 -filter "drivetype=$drivetype" |
 Sort-Object -property DeviceID |
 Format-Table -property DeviceID,
     @{l='FreeSpace(MB)';e={$_.FreeSpace / 1MB -as [int]}},
     @{l='Size(GB';e={$_.Size / 1GB -as [int]}},
     @{l='%Free';e={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Normally, PowerShell ignores anything on a line that follows a `#` symbol, meaning that `#` designates a line as a comment. I've used a `<#  #>` block comment syntax instead, because I had several lines of comments and didn't want to have to start each line with a separate `#` character.

Now I can drop to the normal console host and ask for help by running `help .\Get-DiskInventory` (again, you have to provide a path because this is a script and not a built-in cmdlet). Figure 17.2 shows the results, which proves that PowerShell is reading those comments and creating a standard help display. I can even run `help .\Get-DiskInventory -full` to get full help, including parameter information and my example. Figure 17.3 shows those results.

These special comments, called comment-based help, must appear at the beginning of your script file. There are several keywords in addition to `.DESCRIPTION`, `.SYNOPSIS`, and the others I've used. For a full list, run `help about_comment_based_help` in PowerShell.

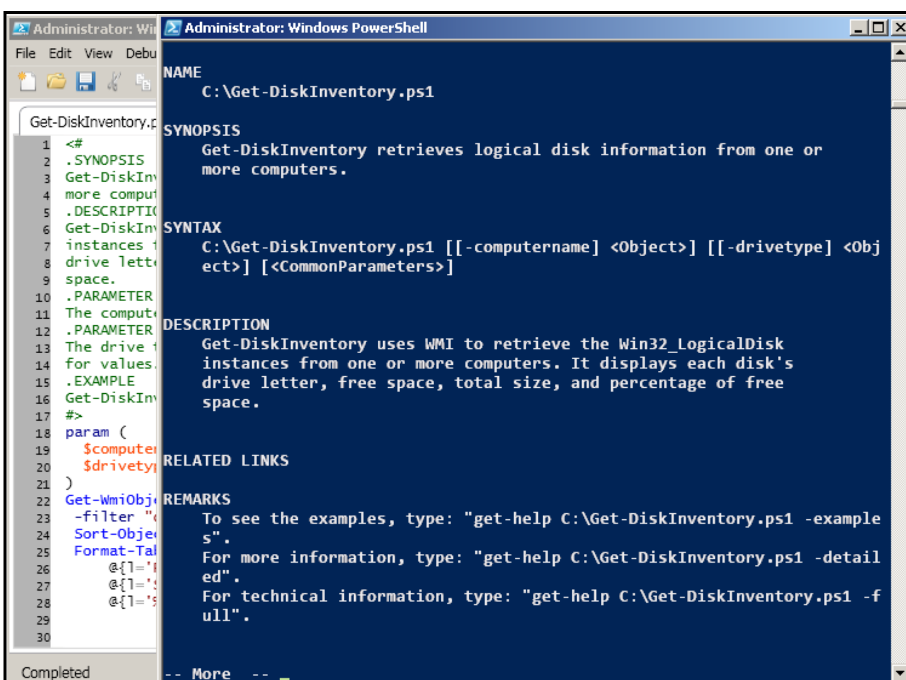



**Figure 17.2 Viewing the help by using the normal help command**

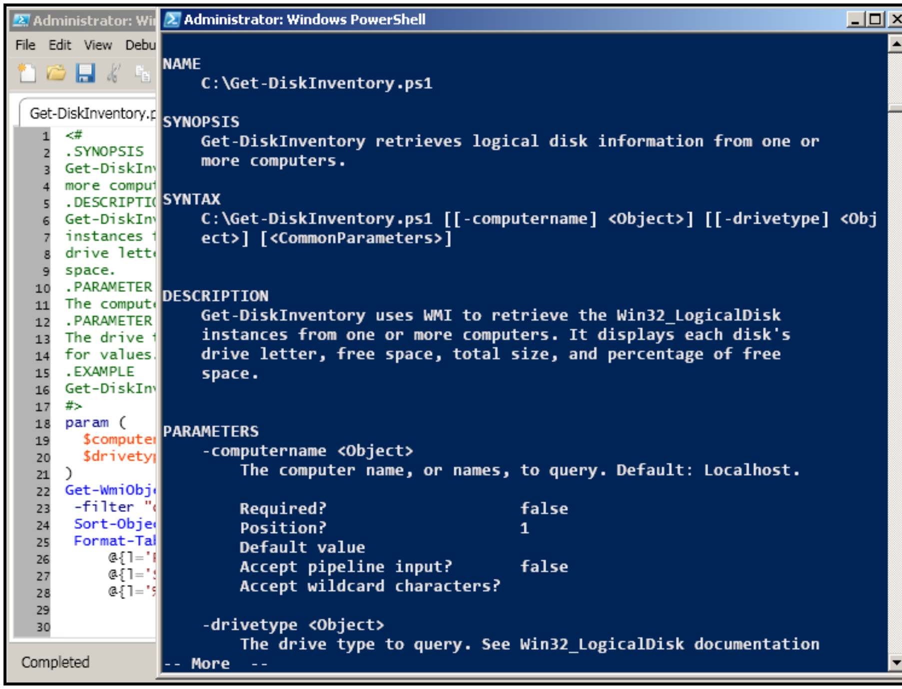



**Figure 17.3 Help options like `-example`, `-detailed`, and `-full` are supported for comment-based help.**

## 17.6 One script, one pipeline

I normally tell folks that anything in a script will run exactly as if you manually typed it into the shell, or if you copied the script to the clipboard and pasted it into the shell. That's not entirely true, though.

Consider this simple script:

```
Get-Process
Get-Service
```

Just two commands. But what happens if you were to type those commands into the shell manually, hitting Return after each?

> **TRY IT NOW** You're going to have to run these commands on your own to see the results; they create fairly long output and it won't fit well within this book or even in a screenshot.

When you run the commands individually, you're creating a new pipeline for each command. At the end of each pipeline, PowerShell looks to see what needs to be formatted, and creates the tables that you undoubtedly saw. The key here is that *each command runs in a separate pipeline*. Figure 17.4 illustrates this: two completely separate commands, two individual pipelines, two formatting processes, and two different-looking sets of results.

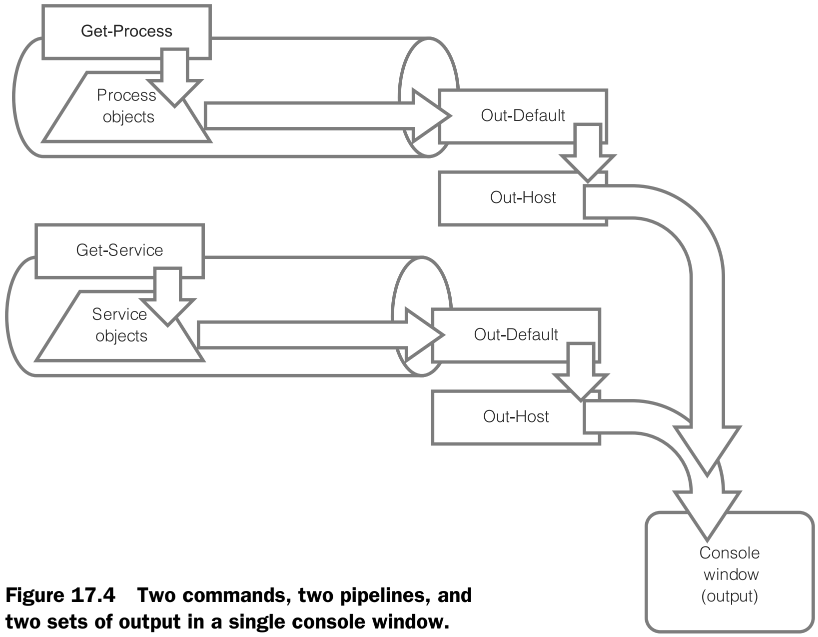



**Figure 17.4 Two commands, two pipelines, and two sets of output in a single console window.**

You may think I'm crazy for taking so much time to explain something that probably seems obvious, but it's important. Here's what happens when you run those two commands individually:

1. You run `Get-Process`.
2. The command places `Process` objects into the pipeline.
3. The pipeline ends in `Out-Default`, which picks up the objects.
4. `Out-Default` passes the objects to `Out-Host`, which calls on the formatting system to produce text output (you learned about this in chapter 8).
5. The text output appears on the screen.
6. You run `Get-Service`.
7. The command places `Service` objects into the pipeline.
8. The pipeline ends in `Out-Default`, which picks up the objects.
9. `Out-Default` passes the objects to `Out-Host`, which calls on the formatting system to produce text output.
10. The text output appears on the screen.

So you're now looking at a screen that contains the results from two commands. I want you to put those two commands into a script file. Name it Test.ps1 or something simple. Before you run the script, though, copy those two commands onto the clipboard. In the ISE, you can highlight both lines of text and press Ctrl-C to get them into the clipboard.

With those commands in the clipboard, go to the PowerShell console host and press Enter. That will paste the commands from the clipboard into the shell. They

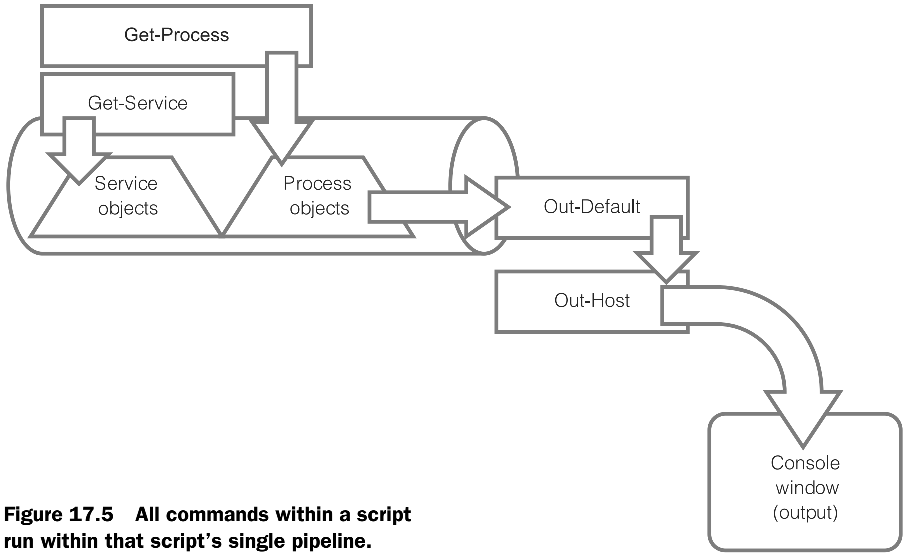



**Figure 17.5 All commands within a script run within that script's single pipeline.**

should execute exactly the same way, because the carriage returns also get pasted. Once again, you're running two distinct commands in two separate pipelines.

Now go back to the ISE and run the script. Different results, right? Why is that?

In PowerShell, every command executes within a single pipeline, and that includes scripts. Within a script, any command that produces pipeline output will be writing to a single pipeline: the one that the script itself is running in. Take a look at figure 17.5.

I'll try to explain what happened:

1. The script runs `Get-Process`.
2. The command places `Process` objects into the pipeline.
3. The script runs `Get-Service`.
4. The command places `Service` objects into the pipeline.
5. The pipeline ends in `Out-Default`, which picks up both kinds of objects.
6. `Out-Default` passes the objects to `Out-Host`, which calls on the formatting system to produce text output.
7. Because the `Process` objects are first, the shell's formatting system selects a format appropriate to processes. That's why they look normal. But then the shell runs into the `Service` objects. It can't produce a whole new table at this point, so it winds up producing a list.
8. The text output appears on the screen.

This different output occurs because the script wrote two kinds of objects to a single pipeline. This is the important difference between putting commands into a script and running them manually: within a script, you only have one pipeline to work with. Normally, your scripts should strive to only output one kind of object, so that PowerShell can produce sensible text output.

## 17.7 A quick look at scope

The last topic we need to visit is *scope*. Scopes are a form of container for certain types of PowerShell elements, primarily aliases, variables, and functions.

The shell itself is the top-level scope and is called the *global scope*. When you run a script, a new scope is created around that script, and it's called the *script scope*. The script scope is subsidiary to the global scope and is said to be a *child* of the global scope, which is the script scope's *parent*. Functions (which you'll learn about in chapter 19) also get their own *private scope*.

Figure 17.6 illustrates these scope relationships, with the global scope containing its children, and those containing their own children, and so forth.

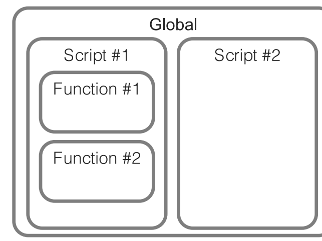



**Figure 17.6 Global, script, and function (private) scopes**

A scope only lasts as long as needed to execute whatever is in the scope. That means the global scope only exists while PowerShell is running, a script scope only exists while that script is running, and so forth. Once whatever it is stops running, the script vanishes, taking everything inside it with it. PowerShell has some very specific—and sometimes confusing—rules for scoped elements like aliases, variables, and functions, but the main rule is this: If you try to access a scoped element, PowerShell sees if it exists within the current scope. If it doesn't, PowerShell sees if it exists in the current scope's parent. It continues going up the relationship tree until it gets to the global scope.

**TRY IT NOW** In order to get the proper results, it's important that you follow these steps carefully and precisely.

Let's see this in action. Follow these steps:

1. Close any PowerShell or PowerShell ISE windows you may have open, so that you can start from scratch.
2. Open a new PowerShell window, and a new PowerShell ISE window.
3. In the ISE, create a script that contains one line: `Write $x`
4. Save the script as C:\Scope.ps1.
5. In the regular PowerShell window, run `C:\Scope`. You shouldn't see any output. When the script ran, a new scope was created for it. The `$x` variable didn't exist in that scope, so PowerShell went to the parent scope—the global scope—to see if `$x` existed there. It didn't exist there, either, so PowerShell decided that `$x` was empty, and wrote that (meaning, nothing) as the output.
6. In the normal PowerShell window, run `$x = 4`. Then, run `C:\Scope` again. This time, you should see 4 as output. The variable `$x` still wasn't defined in the script scope, but PowerShell was able to find it in the global scope, and so the script used that value.
7. In the ISE, add `$x = 10` to the top of the script (before the existing `Write` command), and save the script.

8 In the normal PowerShell window, run `C:\Scope` again. This time, you see 10 as output. That's because `$x` was defined within the script scope, and the shell didn't need to look in the global scope. Now run `$x` in the shell. You'll see 4, proving that the value of `$x` within the script scope didn't affect the value of `$x` within the global scope.

One important concept here is that when a scope defines a variable, alias, or function, that scope loses access to any variables, aliases, or functions having the same name in a parent scope. The locally defined element will always be the one PowerShell uses. For example, if you put `New-Alias Dir Get-Service` into a script, then within that script the alias `Dir` will run `Get-Service` instead of the usual `Get-ChildItem`. (In reality, the shell probably won't let you do that, because it protects the built-in aliases from being redefined.) By defining the alias within the script's scope, you prevent the shell from going to the parent scope and finding the normal, default `Dir`. Of course, the script's redefinition of `Dir` will only last for the execution of that script, and the default `Dir` defined in the global scope will remain unaffected.

It's easy to let this scope stuff confuse you. You can avoid confusion by never relying on anything that's in any scope other than the current one. So before you try to access a variable within a script, make sure you've already assigned it a value within that same scope. Parameters in a `Param()` block are one way to do that, and there are many other ways to put values and objects into a variable.

## 17.8 Lab

The following command is for you to add to a script. You should first identify any elements that should be parameterized, such as the computer name. Your final script should define the parameter, and you should create comment-based help within the script. Run your script to test it, and use the `Help` command to make sure your comment-based help works properly. Don't forget to read the help files referenced within this chapter for more information.

Here's the command:

```
Get-WmiObject -class Win32_OperatingSystem `
 -computername 'localhost' |
 Where-Object { $_.BuildNumber -ge 7600 } |
 Format-Table __SERVER,Caption,BuildNumber,ServicePackMajorVersion,
  @{l='BIOSSerial';e={
   Get-WmiObject -class Win32_BIOS -computername $_.__SERVER |
    Select-Object -expand SerialNumber
   }}
```

## 17.9 Ideas for on your own

Go back through some of the previous chapters and find some commands that you think you might want to run more than once. Chapter 7, which included some examples of Active Directory commands, might be a good choice. Once you've found a command or two, try making them into a parameterized script, and test your script to see if it works. Add comment-based help to explain how the script works, and try viewing the help by using the standard `Help` command.

# 18 Sessions: remote control, with less work

Back in chapter 10, I introduced you to PowerShell's remoting features. In that chapter, you used two primary cmdlets—`Invoke-Command` and `Enter-PSSession`—to access both one-to-many and one-to-one remote control. Each of those cmdlets worked by creating a new remoting connection, doing whatever work you specified, and then closing that connection. There's nothing wrong with that approach, but it can be tiring to have to continually specify computer names, credentials, alternative port numbers, and so on. In this chapter, we'll look at an easier, more reusable way of tackling remoting. You'll also learn about a third way of using remoting that will really come in handy.

## 18.1 Making PowerShell remoting a bit easier

Anytime you need to connect to a remote computer, using either `Invoke-Command` or `Enter-PSSession`, you have to at the very least specify the computer's name (or names, if you're invoking a command on multiple computers). Depending on your environment, you may also have to specify alternative credentials, which means being prompted for a password. You might also need to specify alternative ports or authentication mechanisms, depending upon how remoting is configured in your organization.

None of that is difficult to specify, but it can be tedious to have to do so again and again and again. Fortunately, there's a better way: reusable *sessions*.

## 18.2 Creating and using reusable sessions

A session is a persistent connection between your copy of PowerShell and a remote copy of PowerShell. While the session is active, both your computer and the remote machine devote a small amount of memory and processor time toward maintaining the connection, although there's very little network traffic involved in the connection. PowerShell maintains a list of all the sessions that you've opened, and you can utilize those sessions to invoke commands or to enter a remote shell.

To create a new session, use the `New-PSSession` cmdlet. Specify the computer name (or names), and, if necessary, specify an alternative username, port, authentication mechanism, and so forth. The result will be a session object, which is stored in PowerShell's memory.

```
PS C:\> new-pssession -computername server-r2,server17,dc5
```

To retrieve those sessions, run `Get-PSSession`:

```
PS C:\> get-pssession
```

That works, but I prefer to create the sessions and then immediately store them in a variable. For example, I have three IIS-based web servers that I routinely reconfigure by using `Invoke-Command`. To make that easier, I'll store those sessions in a specific variable:

```
PS C:\> $iis_servers = new-pssession -comp web1,web2,web3
➥ -credential WebAdmin
```

Never forget that those sessions consume resources. If you close the shell, they'll close automatically, but if you're not actively using them, it's a good idea to manually close them even if you're planning to continue using the shell for other tasks.

To close a session, use the `Remove-PSSession` cmdlet. For example, to close just the IIS sessions, use this command:

```
PS C:\> $iis_servers | remove-pssession
```

Or, if you want to close all open sessions, use this command:

```
PS C:\> get-pssession | remove-pssession
```

Easy enough.

Once you get some sessions up and running, what will you do with them? For the next couple of sections, I'll assume that you have created a variable named `$sessions` that contains at least two sessions. I'll use localhost and SERVER-R2; you should specify your own computer names. Using localhost isn't cheating: PowerShell actually starts up a real remoting session with another copy of itself. Keep in mind that this will only work if you've enabled remoting on all computers that you're connecting to, so revisit chapter 10 if you haven't done so.

> **TRY IT NOW** Start following along and running these commands, being sure to use valid computer names. If you only have one computer, use both its name and localhost.

Here's how I'll get my sessions up and running:

```
PS C:\> $sessions = New-PSSession -comp SERVER-R2,localhost
```

Bear in mind that I've already enabled remoting on these computers and that they're all in the same domain. Revisit chapter 10 if you'd like a refresher on enabling remoting.

## 18.3 Using sessions with Enter-PSSession

As you hopefully recall from chapter 10, `Enter-PSSession` is the cmdlet you use to engage a one-to-one remote interactive shell with a single remote computer. Rather than specifying a computer name with the cmdlet, you can specify a single session object. Because my `$sessions` variable has two session objects, I must specify one of them using an index (which you first learned to do in chapter 15):

```
PS C:\> enter-pssession -session $sessions[0]
[server-r2]: PS C:\Users\Administrator\Documents>
```

You can see that my prompt changed to indicate that I'm now controlling a remote computer. `Exit-PSSession` will return me back to my local prompt, but the session will remain open for additional use:

```
[server-r2]: PS C:\Users\Administrator\Documents> exit-pssession
PS C:\>
```

You might have a tough time remembering which index number goes with which computer. In that case, you can take advantage of the properties of a session object. For example, when I pipe my sessions to `Gm`, I get this output:

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

   TypeName: System.Management.Automation.Runspaces.PSSession

Name                   MemberType     Definition
----                   ----------     ----------
Equals                 Method         bool Equals(System.Object obj)
GetHashCode            Method         int GetHashCode()
GetType                Method         type GetType()
ToString               Method         string ToString()
ApplicationPrivateData Property       System.Management.Automation.PSPr...
Availability           Property       System.Management.Automation.Runs...
ComputerName           Property       System.String ComputerName {get;}
ConfigurationName      Property       System.String ConfigurationName {...
Id                     Property       System.Int32 Id {get;}
InstanceId             Property       System.Guid InstanceId {get;}
Name                   Property       System.String Name {get;set;}
Runspace               Property       System.Management.Automation.Runs...
State                  ScriptProperty System.Object State {get=$this.Ru...
```

I can see that the session object has a `ComputerName` property, so I could filter for that session:

```
PS C:\> enter-pssession -session ($sessions |
➥ where { $_.computername -eq 'server-r2' })
[server-r2]: PS C:\Users\Administrator\Documents>
```

That's pretty awkward syntax, though. If you need to use a single session from a variable, and you can't remember which index number is which, it might be easier to forget about using the variable.

Even though you stored your session objects in the variable, they're still also stored in PowerShell's master list of open sessions. That means you can access them by using `Get-PSSession`:

```
PS C:\> enter-pssession -session (get-pssession -computer server-r2)
```

That will retrieve the session having the computer name SERVER-R2 and pass it to the `-session` parameter of `Enter-PSSession`.

When I first figured out that technique, I was impressed with myself, but it also led me to read a bit deeper. I pulled up the full help for `Enter-PSSession` and looked more closely at the `-session` parameter. Here's what I saw:

```
-Session <PSSession>
    Specifies a Windows PowerShell session (PSSession) to use for the
    interactive session. This parameter takes a session object. You ca
    n also use the Name, InstanceID, or ID parameters to specify a PSS
    ession.

    Enter a variable that contains a session object or a command that
    creates or gets a session object, such as a New-PSSession or Get-P
    SSession command. You can also pipe a session object to Enter-PSSe
    ssion. You can submit only one PSSession with this parameter. If y
    ou enter a variable that contains more than one PSSession, the com
    mand fails.

    When you use Exit-PSSession or the EXIT keyword, the interactive s
    ession ends, but the PSSession that you created remains open and a
    vailable for use.

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

If you think back to chapter 7, that pipeline input information near the end is interesting. It tells me that the `-session` parameter can accept, from the pipeline, a `PSSession` object. I know that `Get-PSSession` produces `PSSession` objects, so this syntax should also work:

```
PS C:\> Get-PSSession -ComputerName SERVER-R2 | Enter-PSSession
[server-r2]: PS C:\Users\Administrator\Documents>
```

And it does work! I think that's a much more elegant way to retrieve a single session, even if you've stored all your sessions in a variable.

## 18.4 Using sessions with Invoke-Command

Sessions really show their usefulness with `Invoke-Command`, which you'll remember is used to send a command (or an entire script) to multiple remote computers in parallel. With my sessions in a `$sessions` variable, I can easily target them all with a command:

```
PS C:\> invoke-command -command { get-wmiobject -class win32_process }
➥ -session $sessions
```

Notice that I'm sending a `Get-WmiObject` command to the remote computers. I could have chosen to use `Get-WmiObject`'s own `-computername` parameter, but I didn't do so for four reasons:

- Remoting works over a single, predefined port; WMI doesn't. Remoting is therefore easier to use with computers that are firewalled, because it's easier to make the necessary firewall exceptions. Microsoft's Windows Firewall provides a specific exception for WMI that includes the stateful inspection necessary to make WMI's random port selection (called *endpoint mapping*) work properly, but it can be difficult to manage with some third-party firewall products. With remoting, it's an easy, single port.
- Pulling all of the processes can be labor-intensive, so this way each computer is doing its own share of the work and just sending me the results.
- Remoting operates in parallel, contacting up to 32 computers at once by default. WMI only works sequentially with one computer at a time.
- I can't use my predefined sessions with `Get-WmiObject`, but I can use them with `Invoke-Command`.

The `-session` parameter of `Invoke-Command` can also be fed with a parenthetical command, much as I've done with computer names in previous chapters. For example, this sends a command to every session connected to a computer whose name starts with "loc":

```
PS C:\> invoke-command -command { get-wmiobject -class win32_process }
➥ -session (get-pssession -comp loc*)
```

You might expect that `Invoke-Command` would be able to receive session objects from the pipeline, just as `Enter-PSSession` can. But a glance at the full help for `Invoke-Command` shows that it can't do that particular pipeline trick:

```
-Session <PSSession[]>
    Runs the command in the specified Windows PowerShell sessions (PSS
    essions). Enter a variable that contains the PSSessions or a comma
    nd that creates or gets the PSSessions, such as a New-PSSession or
     Get-PSSession command.

    When you create a PSSession, Windows PowerShell establishes a pers
    istent connection to the remote computer. Use a PSSession to run a
     series of related commands that share data. To run a single comma
    nd or a series of unrelated commands, use the ComputerName paramet
    er.
```

```
To create a PSSession, use the New-PSSession cmdlet. For more info
rmation, see about_PSSessions.

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

Here, the `-session` parameter can only accept pipeline input ByPropertyName, which means I would need to pipe in an object that contained a session object inside a property named `Session`—I can't just pipe in session objects as I did with `Enter-PSSession`. Too bad, but the preceding example of using a parenthetical expression provides the same functionality without too difficult a syntax.

## 18.5 Implicit remoting: importing a session

Implicit remoting, for me, is one of the coolest and most useful—possibly *the* coolest and *the* most useful—feature a command-line interface has ever had, on any operating system, ever. And unfortunately, it's barely documented in PowerShell! Sure, the commands necessary are well documented, but how they come together to form this incredible capability isn't mentioned. Fortunately, I have you covered on this one.

Here's the scenario: We already know that Microsoft is shipping more and more modules and snap-ins with Windows and other products. Sometimes, those modules and snap-ins can't be installed on your local computer for one reason or another. The ActiveDirectory module, which shipped for the first time with Windows Server 2008 R2, is a perfect example: it only exists on Windows Server 2008 R2 and on Windows 7 machines that have the Remote Server Administration Tools (RSAT) installed. What if your computer is running Windows XP or Windows Vista? Are you out of luck? No! You can use implicit remoting!

Here's the entire process, laid out for you in a single example:

```
PS C:\> $session = new-pssession -comp server-r2                  ← 1 Establishes connection
PS C:\> invoke-command -command
➥ { import-module activedirectory }                               ← 2 Load remote module
➥ -session $session
PS C:\> import-pssession -session $session
➥ -module activedirectory                                         ← 3 Import remote commands
➥ -prefix rem

ModuleType Name                       ExportedCommands             ← 4 Reviews temporary local module
---------- ----                       ----------------
Script     tmp_2b9451dc-b973-495d... {Set-ADOrganizationalUnit, Get-ADD...
```

Here's what I do:

1. I start by establishing a session with a remote computer that has the ActiveDirectory module installed 1. That computer needs to be running PowerShell v2 (which Windows Server 2008 R2 does), and it must have remoting enabled.

2 I tell the remote computer to import its local ActiveDirectory module ❷. You could load any module, or even add a PSSnapin. Because the session is still open, the module stays loaded on the remote computer.

3 I then tell my computer to import the commands from that remote session ❸. I only want the commands in the ActiveDirectory module, and when they're imported I want a "rem" prefix added to each command's noun. That way I can keep track of the remote commands more easily.

4 PowerShell creates a temporary module on my computer that represents the remote commands ❹. The commands aren't actually copied over; instead, PowerShell creates shortcuts for them, and those shortcuts point to the remote machine.

Now I can run the ActiveDirectory module commands, or even ask for help. Instead of running `New-ADUser`, I'd run `New-remADUser`, because I added that "rem" prefix to the commands' nouns. The commands will remain available until I either close the shell or close that session with the remote computer. When I open a new shell, I'll have to repeat this process to regain access to the remote commands.

When I run these commands, they don't execute on my local machine. Instead, they're implicitly remoted to the remote computer. It executes them for me and sends the results to my computer.

I can envision a world where we don't ever install administrative tools on our computers again. What a hassle we'd avoid! Today, you have to get tools that can run on your computer's operating system and talk to whatever remote server you're trying to manage—getting everything to match up can be impossible. In the future, you won't do that. You'll use implicit remoting. Servers will offer their management features as another service, via Windows PowerShell.

Now for the bad news: the results brought to your computer through implicit remoting are all deserialized, meaning that the objects' properties are copied into an XML file for transmission across the network. The objects you receive this way don't have any methods. In most cases, that's not a problem, but some modules and snap-ins produce objects that are meant to be used in a more programming-centric style, and those don't lend themselves to implicit remoting. Hopefully, you'll encounter few (if any) objects with this limitation, as a reliance on methods violates some PowerShell design practices. If you do run into such objects, you won't be able to utilize them through implicit remoting.

## 18.6 Lab

To complete this lab, you're going to want to have two computers: one to remote from, and another to remote to. If you only have one computer, use its computer name to remote to it. The remote computer should be running Windows Server 2008 R2. You should get the same essential experience that way.

1. Close all open sessions in your shell.
2. Establish a session to the remote computer. Save the session in a variable named `$session`.
3. Use the `$session` variable to establish a one-to-one remote shell session with the remote computer. Display a list of processes, and then exit.
4. Use the `$session` variable with `Invoke-Command` to get a list of services from the remote computer.
5. Use `Get-PSSession` and `Invoke-Command` to get a list of the 20 most recent Security event log entries from the remote computer.
6. Use `Invoke-Command` and your `$session` variable to load the `ServerManager` module on the remote computer.
7. Import the `ServerManager` module's commands from the remote computer to your computer. Add the prefix "rem" to the imported commands' nouns.
8. Run the imported `Get-WindowsFeature` command.
9. Close the session that's in your `$session` variable.

## 18.7 Ideas for on your own

Take a quick inventory of your environment: what PowerShell-enabled products do you have? Exchange Server? SharePoint Server? VMWare vSphere? System Center Virtual Machine Manager? These and other products all include PowerShell modules or snap-ins, many of which are accessible via PowerShell remoting.

# 19 From command to script to function

Let's say that you've come up with a great command that you not only want to use again and again, but you want to share with your colleagues and co-workers. It's a somewhat complicated command, so you want to put it into a batch file for them. There are also one or two things that need to change each time you run the command, such as a computer name or other parameter, so you want to make it easy for them to provide that information. PowerShell makes it easy to do all of that, and you're going to see how in this chapter.

## 19.1 Turning a command into a reusable tool

If I've said it once, I've said it a million times: you don't need to be a programmer to do amazing things with PowerShell. Given a command that does what you want, you just need to add a little bit of structure—not really programming or scripting code—to modularize it and to start making a reusable tool out of it.

Let's start with two commands. We'll get a computer's name, operating system build number, service pack version, and C: drive free space in megabytes (MB). We can do that right from the shell by using these two commands:

```
Get-WmiObject Win32_OperatingSystem -computer SERVER-R2 |
 Select @{l='ComputerName';e={$_.__SERVER}},
  BuildNumber,ServicePackMajorVersion

Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" -comp SERVER-R2 |
 Select @{l='SysDriveFree(MB)';e={$_.FreeSpace / 1MB -as [int]}}
```

One problem is that these two commands create two completely independent sets of output—try running each of them and you'll see what I mean.

> **TRY IT NOW** Go ahead and try running both commands, but be sure to use a working computer name (or localhost) rather than SERVER-R2. Also try entering both commands into a single script file, using the Windows PowerShell ISE, and running the script. Notice the difference in the output you get between running the commands from the console and running them both in a script.

Another problem is that the computer name is hardcoded into the commands. We want to provide an easy way to substitute a different computer name each time the commands are run.

So at this point we have two problems:

- We want the output combined into a single four-column table.
- We want to be able to easily provide a different computer name, or perhaps multiple computer names, each time the commands are run.

## 19.2 Modularizing: one task, one function

First, I recommend that you try to identify the main tasks that your tool or commands need to perform. Looking at the two commands from the preceding section, I can immediately identify three distinct tasks:

- We need to get a computer name, or several computer names, from someplace.
- We have to retrieve the Win32_OperatingSystem WMI class.
- We have to retrieve the Win32_LogicalDisk WMI class.

You have to be a bit careful about breaking a command into too many tasks, though. A good way to avoid overdoing it is to think about your output. We want a single, combined table containing the operating system information and the disk information. That means those two commands are really part of the same task, because we only want a single piece of output. So that narrows it down to two tasks: retrieving the WMI information, and getting computer names from somewhere.

It's important that we keep those two tasks separate. We need to build two units of work: one unit of work is going to handle the WMI queries and create the output; the other unit of work will provide computer names. In fact, computer names might be provided by whoever is using this command.

Let's stop worrying about the computer names for now, and focus on the other unit of work, which is where the actual functionality is happening. We need to solve the problem of getting all of the output into a single table. One way to do that is to construct our own output table. Listing 19.1 shows that approach, and the output looks like this:

```
ComputerName   BuildNumber   SPVersion   FreeSpace
============   ===========   =========   =========
SERVER-R2   7600   0   50077
```

**Listing 19.1 Trying to get all the output into a single table**

```
$computername = 'SERVER-R2'                          <-- 1 Put computer name in a variable

$os = Get-WmiObject Win32_OperatingSystem -computer $computername |
 Select @{l='ComputerName';e={$_.__SERVER}},
        BuildNumber,ServicePackMajorVersion

$disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
 -computer $computername |
 Select @{l='SysDriveFree';e={$_.FreeSpace / 1MB -as [int]}}

Write-Host "Computer`tBuildNumber`tSPVersion`tFreeSpace"     <-- 2 Output table header
Write-Host "========`t===========`t=========`t========="
Write-Host ("{0}`t{1}`t{2}`t{3}" -f ($os.ComputerName),
 ($os.BuildNumber),($os.servicepackmajorversion),         <-- 3 Use -f operator to make a table row
 ($disk.sysdrivefree))
```

Okay, that output is pretty awful—the columns aren't lining up. But the information we want is there, so let's tackle the formatting later.

In listing 19.1, you can see that I've put the computer name into a variable of its own ①. That separates the computer name from the rest of the task. I'm also using `Write-Host` to create a table header ②, and then using the `-f` operator to format a table row ③. The string that precedes the `-f` operator has placeholders like `{0}` and `{1}`, as well as tab characters (`` `t``). The comma-separated list after the `-f` operator provides the values for each of the four placeholders. This will be our starting point.

## 19.3 Simple and parameterized functions

We want to modularize our code into a function, which is a self-contained, standalone unit of work that we can distribute more easily. An easy way to make a function is to wrap it in a function declaration, as shown here.

**Listing 19.2 Wrapping the code in a function**

```
function Get-ServerInfo {
    $computername = 'SERVER-R2'

    $os = Get-WmiObject Win32_OperatingSystem -computer $computername |
     Select @{l='ComputerName';e={$_.__SERVER}},
            BuildNumber,ServicePackMajorVersion

    $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
     -computer $computername |
     Select @{l='SysDriveFree';e={$_.FreeSpace / 1MB -as [int]}}

    Write-Host "ComputerName`tBuildNumber`tSPVersion`tFreeSpace"
    Write-Host "============`t===========`t=========`t========="
    Write-Host ("{0}`t{1}`t{2}`t{3}" -f ($os.ComputerName),
     ($os.BuildNumber),($os.servicepackmajorversion),
     ($disk.sysdrivefree))
}
```

**TRY IT NOW** You should be able to enter this function into a blank script within the PowerShell ISE. To run the function, just add `Get-ServerInfo` as the last line in the script, underneath the function, and then run the script using the Run toolbar icon (or by pressing F5). That's the same pattern you'll use throughout this chapter: define the function first, and then call the function at the end of the script file.

I've selected a function name that looks like a cmdlet name, using the verb-noun naming convention of a cmdlet. Apart from that, I didn't change anything.

We *need* to change something, though, because we don't want the computer name to always be SERVER-R2. The solution is to change `$computername` from a variable into a parameter, which we can do by adding it to a `Param()` block at the top of the function. The next listing shows this new version of the function.

**Listing 19.3 Parameterizing the function**

```
function Get-ServerInfo {
    param (
        $computername = 'localhost'
    )

    $os = Get-WmiObject Win32_OperatingSystem -computer $computername |
     Select @{l='ComputerName';e={$_.__SERVER}},
            BuildNumber,ServicePackMajorVersion

    $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
     -computer $computername |
     Select @{l='SysDriveFree';e={$_.FreeSpace / 1MB -as [int]}}

    Write-Host "ComputerName`tBuildNumber`tSPVersion`tFreeSpace"
    Write-Host "============`t===========`t=========`t========="
    Write-Host ("{0}`t{1}`t{2}`t{3}" -f ($os.ComputerName),
     ($os.BuildNumber),($os.servicepackmajorversion),
     ($disk.sysdrivefree))
}
```

**Variable changed to parameter**

You'll notice that I set `$computername` equal to `localhost`. That will now serve as a default value. If someone runs this function without specifying a computer name, the function will target localhost, which is usually a safe operation.

The function can be given an alternative computer name in any of these ways:

```
Get-ServerInfo -computername SERVER-R2
Get-ServerInfo -comp SERVER27
Get-ServerInfo WESTDC4
```

**TRY IT NOW** To try any of these examples, add the command to the end of the script file that contains the function. I suggest continuing to work in the PowerShell ISE so that you can follow this pattern throughout the chapter.

In these examples, I used the full parameter name, an abbreviated parameter name, and a positional parameter. All parameters in a `Param()` block are positional by default, meaning that you can pass in values in the order in which the parameters are declared, without specifying the parameter names.

By the way, if you wanted to have a second parameter, you would just separate it from the first one with a comma. There's also a neat trick that allows the default value to prompt the user. That way, if someone runs the function and doesn't specify the parameter, they're prompted for it. That would look something like this:

```
Function Test-This {
  Param(
    $computername = (Read-Host "Enter computer name"),
    $logerrors = $True,
    $logfile
  )
}
```

With the parameter in place, we've completely separated our main functionality from the task of getting computer names.

## 19.4 Returning a value from a function

Now we need to work on the output of our function a bit, because right now it's pretty ugly.

First, though, I want to briefly ignore the function we've been working on and show you one way to output a single value from a function. This is useful in cases where you only need a single value. Here's how you would write the function:

```
function Get-SPVersion {
    param ($computername)
    $os = Get-WmiObject Win32_OperatingSystem -comp $computername
    return ($os.servicepackmajorversion)
}
```

You would run the function, and see the result, as follows:

```
PS C:\> Get-SPVersion server-r2
0
```

You could also capture the function's output into a variable:

```
PS C:\> $version = Get-SPVersion server-r2
PS C:\> $version
0
```

The `return` keyword places a single object (such as an integer, in this example) into the pipeline, and then immediately exits the function. Any code following the `return` keyword won't ever execute.

Our function, however, needs to return more than a single value. It is actually outputting four pieces of information, and we want that information in a table. That means the `return` keyword isn't suitable. Instead, we could continue using `Write-Host`, but it doesn't place anything into the pipeline. That means our function could never pipe its output to another cmdlet like this:

```
PS C:\> Get-ServerInfo | Export-CSV info.csv
```

In order to pipe our output to another cmdlet, we have to place our output into the pipeline, rather than writing it directly to the screen. As you learned in chapter 16, `Write-Output` is the way to do that. This listing shows that modification.

**Listing 19.4 Using `Write-Output` to write to the pipeline**

```
function Get-ServerInfo {
    param (
        $computername = 'localhost'
    )

    $os = Get-WmiObject Win32_OperatingSystem -computer $computername |
     Select @{l='ComputerName';e={$_.__SERVER}},
            BuildNumber,ServicePackMajorVersion

    $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
     -computer $computername |
     Select @{l='SysDriveFree';e={$_.FreeSpace / 1MB -as [int]}}

    Write-Output "ComputerName`tBuildNumber`tSPVersion`tFreeSpace"
    Write-Output "============`t===========`t=========`t========="
    Write-Output ("{0}`t{1}`t{2}`t{3}" -f ($os.ComputerName),
     ($os.BuildNumber),($os.servicepackmajorversion),
     ($disk.sysdrivefree))
}
```

All I've done is swap out `Write-Output` for `Write-Host`. Running this function, I get the same output that we did before, which I'm still not happy with. It looks like this:

```
Get-ServerInfo
ComputerName   Build   Number   SPVersion   FreeSpace
============   =====   ======   =========   =========
SERVER-R2      7600    0        50077
```

But let's try piping the output to a CSV file. Run `Get-ServerInfo | Export-CSV info.csv`, and then open the CSV file in Notepad. This is what I see:

```
#TYPE System.String
"Length"
"44"
"44"
"22"
```

Not what I wanted at all. This isn't going well. I think we need to consider an entirely different way of producing our output.

## 19.5 Returning objects from a function

Here's a little-known secret about PowerShell: it doesn't really like text—it likes *objects*. What we've been doing so far in our function is attempting to format our output as a text table, all on our own. By doing so, we're working against PowerShell's native capabilities, which are making it harder to get the output we want. We need to stop fighting the shell and instead work with it. That means we need to stop trying to output text, and instead output *objects*.

Because we have information from two places—Win32_OperatingSystem and Win32_LogicalDisk—we can't directly output either of the objects we got back from WMI. Instead, we need to create a brand-new, blank object that we can use to combine our four pieces of information. PowerShell provides a blank object type called a `PSObject` for exactly this purpose. We simply need to create one of these, and then add our information to it in the form of properties. Specifically, we'll add our information in the form of a `NoteProperty`, which is a static piece of information.

I'm going to make several changes to our function, as shown in listing 19.5. The good news is that the output is exactly what we want:

```
ComputerName BuildNumber SPVersion SysDriveFree
------------ ----------- --------- ------------
localhost    7600                0            0
```

**Listing 19.5 Outputting objects instead of text**

```
function Get-ServerInfo {
    param (
        $computername = 'localhost'
    )

    $os = Get-WmiObject `
    ➥ Win32_OperatingSystem -computer $computername          ❶ Use simplified WMI query

    $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
     -computer $computername

    $obj = New-Object -TypeName PSObject                      ❷ Create PSObject

    $obj | Add-Member -MemberType NoteProperty `
      -Name ComputerName -Value $computername

    $obj | Add-Member -MemberType NoteProperty `              ❸ Add properties to object
      -Name BuildNumber -Value ($os.BuildNumber)

    $obj | Add-Member -MemberType NoteProperty `
      -Name SPVersion -Value ($os.ServicePackMajorVersion)

    $obj | Add-Member -MemberType NoteProperty `
      -Name SysDriveFree -Value ($disk.free / 1MB -as [int])

    Write-Output $obj                                          ❹ Write object to pipeline
}

Get-ServerInfo | Format-Table -auto                            ❺ Call function
```

In this listing, I've started by simplifying the WMI queries ❶. There's no need to create those custom columns by using `Select-Object`. We're going to be creating and outputting a whole new object, so we can do those customizations right on that new object. After querying the information, I create the new `PSObject` and put it in a variable, `$obj` ❷. To add information to that object, I pipe the object to `Add-Member` four times ❸. Each time, I specify that I'm adding a `NoteProperty`, give a property name, and provide the value for that property. Note that PowerShell isn't usually case-sensitive, but it will preserve whatever case I use, so I've taken care to make sure that the property names are typed with capital letters so that they look nice in the final output. After adding all four properties, I write the final object to the pipeline ❹. This listing also shows the command I used to call the function ❺. You can see that I've piped it to `Format-Table` to ensure I get the output format that I want.

This new function is infinitely flexible, because it outputs objects instead of text. For example, all of these examples are legitimate ways of using the function:

```
Get-ServerInfo | Format-Table -auto
Get-ServerInfo -comp Server-R2 | Export-CSV info.csv
Get-ServerInfo -comp localhost | ConvertTo-HTML | Out-File info.html
```

**TRY IT NOW** You can add all three of these commands to the end of listing 19.5, and then run the complete script to see the results yourself. Be sure to examine the resulting CSV and HTML files.

That last command is my attempt at creating a CSV file again. This time, the results in the CSV file are much better:

```
#TYPE System.Management.Automation.PSCustomObject
"ComputerName","BuildNumber","SPVersion","SysDriveFree"
"Server-R2","7600","0","0"
```

We're definitely on the right path, and here are the two keys to staying there:

- *Break tasks down*—We separated how we get the computer name from the actual working code. All of the working code went into a single function, because we want a single unified piece of output.
- *Output objects*—Always have functions return either a single value (using the `return` keyword), or output objects. By outputting objects, you can pipe the function's output to many other cmdlets to format, convert, filter, sort, and so forth.

You're going to use this object-output technique many more times in the following chapters, so be sure you've taken the time to enter and run these examples, and that you understand what they're doing.

## 19.6 Lab

Create a function of your own that combines information from Win32_OperatingSystem, Win32_BIOS, and Win32_ComputerSystem. Your function's output should include the following:

- The operating system version name (such as "Windows Server 2008 R2")
- The domain that the computer belongs to

- The computer's DNS host name
- The BIOS serial number

Make sure that your function can pipe its output to other cmdlets, such as `Format-List`, `Format-Table`, `Export-CSV`, and `ConvertTo-HTML`.

## 19.7 Ideas for on your own

Spend some time thinking about what other pieces of information you would like to combine into a single piece of output. Operating system version and BIOS serial number? Computer system details and network adapter configuration settings? By creating and outputting your own custom objects, you can combine as much information as you want into a single entity.

# 20 Adding logic and loops

Up to this point, I don't consider anything that we've done so far to be "scripting." It depends on your definition of the word, of course, but to me *scripting* is a kind of programming, with formal constructs that define logic, repetition, and so forth. You can do a lot in PowerShell without that stuff. But the time will come when you *will* need to write a script that can make logical decisions, and you'll start to move beyond running commands and moving into simple scripts.

The goal in this chapter is to let you experience some of PowerShell's major scripting constructs for logic and repetition, so that you'll be prepared to use these elements when the time comes.

## 20.1 Automating complex, multi-step processes

I typically find a need for these constructs when I'm automating more complex, multistep processes. For example, consider a script that provisions a new user: you need to create an Active Directory account, add the user to some groups, create a mailbox, create a home directory on a file server, and so on. Those processes often involve questions, with branching logic: Should the user belong to such-and-such a domain user group? Should they have access to certain files? Each question leads to a slightly different course of action. In some cases, certain operations may have to be done over and over, such as adding a user to several groups, or perhaps granting them permissions over several folders or files.

## 20.2 Now we're "scripting"

I admit that we've been creeping toward actual scripting for a while now. The previous chapter, for example, introduced some structures that would normally only live within a .PS1 script file, and that you'd probably never type directly on the command line. But I mostly think of things like `Param()` blocks as window dressing. They don't do anything, but they do provide some structure and definition to our commands.

Once we start adding logic and repetition, I'll admit that we've formally moved into the world of scripting. That's not a bad thing: this isn't going to be the type of programming you'd do in Visual Studio (although PowerShell can certainly accommodate pretty intense, complex scripts). We're going to keep it simple, using these scripting constructs primarily to add a little intelligence to a batch of commands.

**Above and beyond**

This chapter will introduce you to all but one of PowerShell's formal scripting constructs. The missing construct is the `Do` loop, which also uses the keywords `While` and `Until` in certain scenarios. You can learn more about them in PowerShell's help: run `help about*` to get a list of help topics, and look for `Do`, `While`, and `Until`.

Why aren't they covered in this chapter? Simple: for most administrative scripts, you won't need them. As you start to progress into more advanced scripts, you can familiarize yourself with them on your own, and use them if necessary. They're generally used to repeat some set of commands over and over until a certain condition is either true or false.

## 20.3 *The If construct*

First up is the scripting construct that you'll probably use the most: `If`. A basic `If` construct looks like this:

```
If ($process.pm -gt 10000) {
    Write-Host "This is a large process"
}
```

There are just a couple of important things to note:

- As with most of PowerShell, the `If` keyword isn't case-sensitive. You can use `if` or `IF` or even `iF`.
- The parentheses contain an expression of some kind. This has to evaluate to either `True` or `False` (or, to use the PowerShell values, `$True` or `$False`).
- After the parentheses, you open the construct by using a curly brace. You complete the construct with a closing brace.
- Most people indent the commands within the construct, so that it's easier to visually distinguish the commands that are inside the construct.

Here's another way to format this:

```
If ($process.pm -gt 10000)
{
    Write-Host "This is a large process"
}
```

The only difference is where I put the opening curly brace. PowerShell doesn't care, but this method does make it a bit easier to quickly distinguish the commands inside the construct, and to make sure that you've properly closed the construct. I tend to use the first formatting style because it takes up less room on-screen in a class, and it takes up fewer lines in a book like this.

I find that administrators who are neat and consistent about formatting their constructs typically have to spend less time debugging their scripts, so there's another benefit of properly indenting the commands and so forth.

Sometimes, you may need to check multiple potential conditions. An `ElseIf` allows you to do so:

```
If ($service.name -eq 'BITS') {
    Write-Host 'This is the transfer service'
} elseif ($service.name -eq 'Spooler') {
    Write-Host 'This is the print spooler'
} elseif ($service.name -eq 'W32Time') {
    Write-Host 'This is the time service'
}
```

You can have as many `ElseIf` sections as you want, and each one gets its own conditional expression in parentheses. PowerShell will review these in order, and it will execute only the first one whose expression evaluates to `True`. Once it finds one, it won't evaluate or consider any of the remaining options.

The last permutation of this construct is to add a kind of catch-all that will execute if no preceding condition has been `True`:

```
If ($service.name -eq 'BITS') {
    Write-Host 'This is the transfer service'
} elseif ($service.name -eq 'Spooler') {
    Write-Host 'This is the print spooler'
} elseif ($service.name -eq 'W32Time') {
    Write-Host 'This is the time service'
} else {
    Write-Host 'This is an unknown service'
}
```

The `Else` block comes last, and it will execute only if none of the preceding `If` or `ElseIf` expressions evaluated to `True`. You can use `Else` even if you aren't using any `ElseIf` blocks.

The parenthetical expressions used with `If` and `ElseIf` will often contain a comparison operator, because comparisons are usually an easy way to get a `True` or `False` result. But that isn't always the case. If you have a property or variable that already contains `$True` or `$False`, you don't need a comparison at all. For example, consider this snippet:

```
$processes = Get-Process
if ($processes[0].responding -eq $True) {
    Write-Host 'The first process is responding'
}
```

The Responding property of a process always contains either $True or $False, so there's no need to actually compare it to $True or $False. You could rewrite this as follows:

```
$processes = Get-Process
if ($processes[0].responding) {
    Write-Host 'The first process is responding'
}
```

This is a much more common way of handling the situation. Remember, all you care about is that the interior of the parentheses boils down to $True or $False in some fashion. In this case, because the Responding property already provides one of those two values, you don't need to do any more work.

Here's a quick tip: I learned about the Responding property by running Get-Process | Gm. In the list, I saw Responding and wondered what kind of information it contained. Would it be a 0 or 1? A Yes or No? Something else? So I ran Get-Process | Format-List *, which displayed all of the processes' properties *and their values*. That output showed me that Responding contained True for almost all of my processes, so I logically assumed that False was also a possibility. I encourage you to use this same technique to discover what's inside the properties of other objects you work with.

## 20.4 The Switch construct

The Switch construct acts as a specialized kind of logical comparison. You start with a single variable or property, and you ask the shell to compare its contents to a wide range of possible values. The shell will execute a block of commands for each match that it finds.

Here's an example that translates a numeric printer status code into a human-readable status message:

```
Switch ($printer.status) {
    1075 {
        Write-Host 'Printer jammed.'
    }
    1842 {
        Write-Host 'Toner needed.'
    }
    1167 {
        Write-Host 'Overheating.'
    }
    4422 {
        Write-Host 'Out of paper.'
    }
    'OK' {
        Write-Host 'Operating normally.'
    }
    Default {
        Write-Host 'Status unknown.'
    }
}
```

The `Default` block will execute only if none of the prior blocks have executed. But, as shown here, all possible matches will all execute. That's different than the `If` construct, which only executes the first condition that's `True`.

In this particular example, it's probably impossible for the `Status` property to contain both `1075` and `OK`, so we don't need to worry about multiple matches occurring. But here's a variation of `Switch`:

```
Switch -wildcard ($computername) {
    '*DC*' {
        Write-Host 'Domain Controller'
    }
    '*WEST*' {
        Write-Host 'West Coast'
    }
    '*BK*' {
        Write-Host 'Backup'
    }
}
```

In this case, if `$computername` contains `WESTDCBK`, we'd get "Domain Controller," "West Coast," and "Backup" as our output. That might be desirable, and in this particular scenario I think that's what I'd want. In other situations, however, you might only want the first matching condition to execute. In those cases, use the `Break` keyword within one of the conditional blocks.

`Break` exits the entire construct immediately (it will exit anything except an `If` construct), preventing further potential matches from being considered. Here's an example:

```
Switch -wildcard ($jobtitle) {
    '*Executive*' {
        Write-Host 'Is an executive'
        break
    }
    '*Jan*' {
        Write-Host 'In janitorial'
        break
    }
    '*Manager*' {
        Write-Host 'Is a manager'
        break
    }
}
```

If `$jobtitle` contains `Janitorial Manager`, our only output would be "In janitorial."

The `Switch` construct has a few other tricks it can perform, including evaluating regular expressions (which aren't covered in this book). For more information, run `help about_switch` in the shell.

## 20.5 The For construct

The `For` construct is a loop that's intended to repeat a given block of commands a specific number of times. Here's an example—see if you can predict what its output would be:

```
For ($i = 0; $i -lt 10; $i++) {
    Write-Host $i
}
```

This one can be hard to figure out if you're not familiar with the C-style construction. Here's a cheat sheet:

- The first element is a starting point, where I'm setting a counter variable to `0`.
- The second element is the condition that will keep the loop going. Here, so long as `$i` is less than `10`, the loop will repeat.
- The last element is what to do after each time through the loop. Here, I'm incrementing `$i` by `1`. I could also have typed `$i = $i + 1`, which is a bit easier to figure out, but it takes longer to type than `$i++`.

Can you figure out what the output would be, without actually running that snippet? It would display 0 through 9, and then stop. Once `$i` is no longer less than 10, the loop exits. `$i` will actually contain 10 after the completion of the loop, because it's the fact that `$i` contains 10 that made `$i` no longer less than 10, ending the loop.

## 20.6 The ForEach construct

`ForEach` can be one of the most useful constructs, but it's also the one that I see misused the most. For someone with a programming or VBScript background, `ForEach` can be very familiar and compelling, but it isn't always needed.

Here's what it looks like:

```
$services = Get-Service
ForEach ($service in $services) {
    Write-Host $service.Name
}
```

This script starts by getting a bunch of services, using `Get-Service`, and storing them in the `$services` variable. In the `ForEach` construct, I'm asking it to enumerate (to go through one at a time) all of the services. Each time the loop repeats, the next service will be placed into the `$service` variable, which I made up for just that purpose. There's no need for me to do anything with `$service` ahead of time; by using it in this fashion, I've told PowerShell all it needs to know about what I'm trying to do. The `in` keyword is crucial: the variable *before* the `in` keyword will contain one object at a time; the variable *after* the `in` keyword contains all of the objects I want to work with. Within the construct, I use the one-at-a-time variable (`$service`, in this case) to write the services' names.

ForEach may seem familiar to you, because we've used the ForEach-Object cmdlet (and perhaps you've used that cmdlet's foreach alias) in previous chapters. The construct works much the same as the cmdlet: both go through a collection of objects one at a time. With the construct, you get to define the variable that contains one object at a time (I used $service in the preceding example); with the cmdlet, PowerShell forces you to use the $_ placeholder. For example, I could rewrite the previous example like this:

```
Get-Service | ForEach-Object { Write-Host $_.Name }
```

When you use the construct, you use the in keyword, as I did in my ForEach example. When you use the cmdlet, you don't need to use the in keyword because PowerShell automatically enumerates into the built-in $_ placeholder.

## 20.7 Why scripting isn't always necessary

ForEach is an excellent example of why scripting like this isn't always necessary in PowerShell, even though PowerShell will let you do it. That previous example could have been accomplished more easily in the pipeline:

```
Get-Service | Select Name
```

That will produce new objects that have only a Name property. If you wanted to get the actual names as simple string values, you could do this:

```
Get-Service | Select -expand Name
```

Both of those options involve a lot less typing. The point here is that using ForEach is often (but not always) an indicator that you're taking a scripting approach rather than a pure PowerShell approach. You won't get in trouble for taking a scripting approach, but it often requires a lot more typing—and a script—than using a couple of commands. There are only two times when I find myself legitimately using ForEach:

- When there's no cmdlet capable of doing what I need to a bunch of objects at once. This most often happens when I need to execute a method against a bunch of objects, and there isn't a cmdlet that can perform the equivalent task.
- When I need to manually "unwind" a bunch of objects and send them off, one at a time, to a custom function that I've written, which can only work with one at a time. You'll see an example of this in the next chapter.

If you have some scripting or programming in your background, and you want to try to force yourself to take a more "pure PowerShell" approach, use ForEach as a cue. When you find yourself using it, see if there isn't an easier way. For example, instead of this,

```
$processes = Get-Process
ForEach ($process in $processes) {
    If ($process.name -eq 'notepad') {
        $process.kill()
    }
}
```

you could just do this,

```
Get-Process | Where-Object { $_.Name -eq 'notepad' } | Stop-Process
```

or better yet, this,

```
Get-Process -name notepad | Stop-Process
```

or best of all, this:

```
Stop-Process -name notepad
```

These all accomplish the same thing, but the command-oriented way takes a lot less typing (and to me, is easier to read and figure out) than the scripting-oriented way.

## 20.8 Lab

For this lab, you'll probably want to work within the PowerShell ISE. That will make it easier to enter multiline scripts and commands, and it'll make it easier to edit if you make any mistakes or want to make a change.

1. Create a script that uses `Read-Host` to prompt for a remote computer name. If the computer name is localhost, then don't do anything. Otherwise, query the Win32_OperatingSystem WMI class from the specified computer.
2. Create a script that queries the Win32_LogicalDisk WMI class from the local computer. Use a `ForEach` loop to enumerate the instances returned by the query. Within the `ForEach` loop, display the `DeviceID` property. Then display a text description of the `DriveType` property by using a `Switch` construct. For example, if the `DriveType` is `3`, display "Fixed Disk." Use a search engine to search for "Win32_LogicalDisk," and you'll locate the documentation page for that WMI class. The documentation page will display the possible values, and meanings, of the `DriveType` property.

# 21 Creating your own "cmdlets" and modules

At the end of chapter 19, you saw how to make a function (in listing 19.5) that output custom objects to the pipeline. Mastering that kind of output is a key to becoming a PowerShell guru, but there's also the question of input.

In chapter 19, we passed input to the function by means of a parameter. In this chapter, we're going to look at some other means of getting input into the function. By combining different input techniques with what you already know about producing output, you'll find that you can create a tool that behaves almost exactly like a PowerShell cmdlet!

## 21.1 Turning a reusable tool into a full-fledged cmdlet

As I said, the function in listing 19.5 accepted input primarily through a parameter. In order to make a tool like that more useful, it would be nice if we could pass in multiple pieces of input (the function in listing 19.5 only worked with a single computer name, for example), and pass them in either using a parameter or from the pipeline. That would give us a fully reusable tool that looks and works much like a cmdlet. Ideally, we could even have the shell do some input validation for us, such as marking a parameter as mandatory and automatically prompting the user if it wasn't provided.

To get you to that point, I'm going to take a slightly roundabout path. There are three broad kinds of functions you can write in the shell. Chapter 19 covered one of them—a simple parameterized function (I guess a non-parameterized function could be considered to be an even simpler, fourth type, but I don't write many of those myself). I'll start by showing you a second type, which accepts pipeline input instead.

## 21.2 Functions that work in the pipeline

The next type of function I'll introduce you to is called a *pipeline function*, or *filtering function*. If a regular parameterized function is distinguished by its ability to accept input only through parameters, then a filtering function has these distinguishing characteristics:

- You can accept one kind of information through the pipeline. This might be computer names, processes, or any other single kind of information.
- Whatever you accept through the pipeline can come as a single object, or multiple objects can be piped in. You'll write one (or many) commands that execute against each piped-in object, no matter how many there are.
- You can designate additional parameters for other input elements. The values provided to these parameters will be used for each execution of your commands.

That'll all probably make more sense with an example. We'll start with the same function in listing 19.5, but I'll dress it up slightly to make it a filtering function, shown next.

**Listing 21.1 A filtering function**

```
function Get-ServerInfo {
    BEGIN {}                                  1 Runs first
    PROCESS {                                 2 Defines PROCESS block
        $computername = $_                    3 Uses $_ placeholder

        $os = Get-WmiObject Win32_OperatingSystem -computer $computername

        $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
         -computer $computername

        $obj = New-Object -TypeName PSObject

        $obj | Add-Member -MemberType NoteProperty `
          -Name ComputerName -Value $computername

        $obj | Add-Member -MemberType NoteProperty `
          -Name BuildNumber -Value ($os.BuildNumber)

        $obj | Add-Member -MemberType NoteProperty `
          -Name SPVersion -Value ($os.ServicePackMajorVersion)

        $obj | Add-Member -MemberType NoteProperty `
          -Name SysDriveFree -Value ($disk.free / 1MB -as [int])

        Write-Output $obj
    }
    END {}                                    4 Runs last
}

Get-Content names.txt | Get-ServerInfo | Format-Table -auto
```

This function isn't terribly different from the original one (refer back to listing 19.5 to see that one). I've added a `BEGIN` block ❶ and, at the end of the function, an `END` block ❹. Whatever's inside of the `BEGIN` block will execute the first time this function is called in the pipeline; the `END` block will execute when the function is almost finished. As you can see, I don't put any code in these, so nothing will happen during those two stages. I could omit `BEGIN` and `END` entirely, but I like to include them to keep the structure consistent across all of my functions.

The `PROCESS` script block is where the magic happens ❷. This block will execute one time for each object that's piped into the function (if you don't pipe in any input, the `PROCESS` block will execute once). This script expects computer names to be piped in, so if you pipe in four names, the `PROCESS` block will run four times. Each time, the `$_` placeholder ❸ will be automatically populated with a new object from the pipeline. But rather than utilizing `$_` directly, I've copied its object into the `$computername` variable. Doing so has two advantages: First, my commands were already using `$computername`, so continuing to use it means less work for me. Second, the variable name is clearer than `$_`, making it easier for me to keep track of what the variable is supposed to contain. There's a third, overlooked advantage: `$_` will be repopulated if an error occurs, so by copying it to `$computername` now, I won't lose the initial value.

The last line of the script shows how you would execute this function: pipe a bunch of string objects to it. So long as those objects are computer names, everything should work fine. Another way to execute it would be this:

```
Get-ADComputer -filter * | Select -expand Name | Get-ServerInfo
```

That will retrieve all computers from Active Directory, expand their `Name` properties into simple `String` objects, and pipe those `String` objects to the `Get-ServerInfo` function.

You could also add additional parameters, by including a standard `Param()` block right at the top of the function, before the `BEGIN` block. Whatever values are passed to those parameters will hold the same values each time the `PROCESS` block executes.

That brings up an interesting problem: what if you want the cmdlet to accept computer names either from the pipeline *or* from a parameter? In other words, you want both of these to work:

```
Get-Content names.txt | Get-ServerInfo
Get-ServerInfo -computername (Get-Content names.txt)
```

Right now, the function won't do that, because we don't have a `-computername` parameter defined. The only input expected is that coming from the pipeline. So let's add a parameter, as shown next.

**Listing 21.2 Adding a parameter to a filtering function**

```
function Get-ServerInfo {
    param (
        [string]$computername
    )
```

```
    BEGIN {}
    PROCESS {
        $computername = $_

        $os = Get-WmiObject Win32_OperatingSystem -computer $computername

        $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
         -computer $computername

        $obj = New-Object -TypeName PSObject

        $obj | Add-Member -MemberType NoteProperty `
          -Name ComputerName -Value $computername

        $obj | Add-Member -MemberType NoteProperty `
          -Name BuildNumber -Value ($os.BuildNumber)

        $obj | Add-Member -MemberType NoteProperty `
          -Name SPVersion -Value ($os.ServicePackMajorVersion)

        $obj | Add-Member -MemberType NoteProperty `
          -Name SysDriveFree -Value ($disk.free / 1MB -as [int])

        Write-Output $obj
    }
    END {}
}

Get-Content names.txt | Get-ServerInfo | Format-Table -auto
```

Now, however, we run into a problem. The original way of running the command—which is included at the bottom of the script listing—will continue to work. It produces output that looks like this (assuming names.txt contained SERVER-R2 and localhost):

```
ComputerName   BuildNumber   SPVersion   SysDriveFree
------------   -----------   ---------   ------------
server-r2      7600                  0              0
localhost      7600                  0              0
```

But the other way of running the function doesn't work:

```
Get-ServerInfo -computername (Get-Content c:\names.txt)
Get-WmiObject : Cannot validate argument on parameter 'ComputerName'. The
     argum
ent is null or empty. Supply an argument that is not null or empty and then
     try
 the command again.
At line:9 char:60
+         $os = Get-WmiObject Win32_OperatingSystem -computer <<<<
     $computerna
me
    + CategoryInfo          : InvalidData: (:) [Get-WmiObject],
     ParameterBindi
   ngValidationException
```

```
    + FullyQualifiedErrorId :
     ParameterArgumentValidationError,Microsoft.Power
   Shell.Commands.GetWmiObjectCommand

Get-WmiObject : Cannot validate argument on parameter 'ComputerName'. The
     argum
ent is null or empty. Supply an argument that is not null or empty and then
     try
 the command again.
At line:12 char:19
+          -computer <<<<  $computername
    + CategoryInfo          : InvalidData: (:) [Get-WmiObject],
     ParameterBindi
   ngValidationException
    + FullyQualifiedErrorId :
     ParameterArgumentValidationError,Microsoft.Power
   Shell.Commands.GetWmiObjectCommand


ComputerName        BuildNumber         SPVersion          SysDriveFree
------------        -----------         ---------          ------------
```

Ouch. Lots of ugly errors. Here's the problem: all of the code in the function lives within the `PROCESS` block. We're taking the computer name from the `$_` placeholder, which is populated with an object from the pipeline input. Except that *we didn't pipe anything in*, so the `PROCESS` block only executes once, and `$_` never contains anything, so `$computername` never contains anything, so nothing works. Sigh.

You need to break the function into two pieces. That way, the main working part can stand alone and can be used whether you're getting input from the pipeline or from a parameter. That part will be a behind-the-scenes function that won't be called directly—what I call a *worker function*. The second part will be the *public function* that you want people to actually use. Its whole job will be to figure out where input is coming from, and then to pass one computer name at a time to the worker function.

You need to keep a couple of things in mind:

- When input comes from the pipeline, the shell will enumerate through the objects automatically, allowing you to work with one at a time. That's what listing 21.1 did. You can pass those objects, as they're processed, to the worker function.
- When input comes from a parameter, you may have either one object or many objects, but the `PROCESS` script block will only execute once regardless. So you'll have to manually enumerate, or *unwind*, the parameter so that you can get to each object, one at a time.

Here's the trick: PowerShell has a built-in variable called `$PSBoundParameters`, and it contains each parameter that was manually specified. It has a `ContainsKey()` method that will let you test to see if a particular parameter was used or not.

Here we see the revised script, this time with two functions.

**Listing 21.3 Breaking the function into two parts**

```
function GetServerInfoWork {
    param([string]$computername)
    $os = Get-WmiObject Win32_OperatingSystem -computer $computername

    $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
     -computer $computername

    $obj = New-Object -TypeName PSObject

    $obj | Add-Member -MemberType NoteProperty `
      -Name ComputerName -Value $computername

    $obj | Add-Member -MemberType NoteProperty `
      -Name BuildNumber -Value ($os.BuildNumber)

    $obj | Add-Member -MemberType NoteProperty `
      -Name SPVersion -Value ($os.ServicePackMajorVersion)

    $obj | Add-Member -MemberType NoteProperty `
      -Name SysDriveFree -Value ($disk.free / 1MB -as [int])

    Write-Output $obj
}
function Get-ServerInfo {
    param (
        [string[]]$computername
    )
    BEGIN {
        $usedParameter = $False
        if ($PSBoundParameters.ContainsKey('computername')) {
            $usedParameter = $True
        }
    }
    PROCESS {
        if ($usedParameter) {
            foreach ($computer in $computername) {
                GetServerInfoWork -computername $computer
            }
        } else {
            GetServerInfoWork -computername $_
        }
    }
    END {}
}

Get-ServerInfo -verbose -computername (Get-Content c:\names.txt)
Get-Content c:\names.txt | Get-ServerInfo | Format-Table -auto
```

1 Define worker function
2 Define single-string parameter
3 Define array parameter
4 See if parameter is used
5 Check for pipeline versus parameter input
6 Unwind parameter input
7 Pass along pipeline input

**TRY IT NOW** Note that you can't run this script as-is. You have to comment out one of the two last lines. Try running this with the last line commented out, and then try running it a second time with only the next-to-last line commented out.

In this script, I started by pulling most of the function code into a worker function 1. You can see that I didn't use the normal cmdlet-style naming convention for this one, because I don't expect people to call it directly. I declared a parameter for it 2, and set it up to accept a single string. The rest of the function is unchanged—I simply cut and pasted it from the old `Get-ServerInfo`.

In the revised public function, I declared the parameter to accept multiple strings—that's what the `[string[]]` denotes 3. That way, the parameter will accept one string, or several. In the `BEGIN` block, which executes first, I want to see if the input is coming from the pipeline or via the parameter 4. I start by assuming that the input came from the pipeline, setting `$usedParameter` to `$False`. Then I test the `$PSBoundParameters` variable, and if it does indeed contain the `computername` key, then I know that the `-computerName` parameter was used, so I set `$usedParameter` to `$True`. The `$usedParameter` variable is valid throughout the function; even though it's created in the `BEGIN` block, it will still be accessible in the `PROCESS` and `END` blocks.

In the `PROCESS` block, I check that variable to see what to do 5. Remember that `PROCESS` will execute once if there is no pipeline input, so I use a `ForEach` loop to enumerate the parameter input, passing each object to the worker function one at a time 6. On the other hand, if the input came from the pipeline, the `PROCESS` block is already handling the enumeration, so I let it do its job and pass the `$_` placeholder's contents to the worker function 7.

This may seem like a complex structure, but you can really use this as a template for your own scripts. In fact, I use this exact script as a template all the time. The public function's structure doesn't change much—I pretty much just change the parameter name to suit. The worker function changes a lot, of course, because that's where the actual commands are being run.

Note that it's completely valid to have additional parameters. Listing 21.4 shows an example, where I've added a `-logfile` parameter. Because I'm not expecting that one to have pipeline input, it's much easier to deal with. I just need to make sure I pass the parameter to the worker function, and that the worker function is set up to deal with it. You can see that I'm not using the parameter within the worker function; this is just an example of how you'd do so in your own scripts.

**Listing 21.4 Adding a second parameter**

```
function GetServerInfoWork {
    param([string]$computername,[string]$logfile)
    $os = Get-WmiObject Win32_OperatingSystem -computer $computername

    $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
     -computer $computername

    # use $logfile to get the value from the
    # -logfile parameter

    $obj = New-Object -TypeName PSObject
```

```
    $obj | Add-Member -MemberType NoteProperty `
      -Name ComputerName -Value $computername

    $obj | Add-Member -MemberType NoteProperty `
      -Name BuildNumber -Value ($os.BuildNumber)

    $obj | Add-Member -MemberType NoteProperty `
      -Name SPVersion -Value ($os.ServicePackMajorVersion)

    $obj | Add-Member -MemberType NoteProperty `
      -Name SysDriveFree -Value ($disk.free / 1MB -as [int])

    Write-Output $obj
}
function Get-ServerInfo {
    param (
        [string[]]$computername,
        [string]$logfile
    )
    BEGIN {
        $usedParameter = $False
        if ($PSBoundParameters.ContainsKey('computername')) {
            $usedParameter = $True
        }
    }
    PROCESS {
        if ($usedParameter) {
            foreach ($computer in $computername) {
                GetServerInfoWork -computername $computer `
                 -logfile $logfile
            }
        } else {
            GetServerInfoWork -computername $_ `
             -logfile $logfile
        }
    }
    END {}
}

#Get-ServerInfo -verbose -computername (Get-Content c:\names.txt)
Get-Content c:\names.txt | Get-ServerInfo -logfile test.txt |
Format-Table -auto
```

**TRY IT NOW** You can see that I've commented out one of the script's final lines, so that I'm only running one test at a time.

## 21.3 Functions that look like cmdlets

We're coming very close to creating a function that looks and works, for almost all purposes, like a real cmdlet. About the only thing we're missing is declarative pipeline input.

In the previous example, I checked to see if the `-computerName` parameter was used. If it was, I used the parameter, and if it wasn't, I used `$_` instead. With declarative pipeline input, you can have the shell automatically attach the pipeline input to the `-computerName` (or whatever) parameter, leaving you one less thing to deal with. At the same time, you can ask the shell to do a lot of parameter input validation, like making sure mandatory parameters are specified. All of this mainly involves messing around with the `Param()` block to create a more formal kind of parameter declaration—a *cmdlet binding*–style of declaration, to be exact. Finally, you don't have to use `PSBoundParameters`: your input will always be in the variable defined for the parameter.

The `PROCESS` script block will execute at least once, so you can simplify things a lot. If input comes as a parameter, `PROCESS` will execute once and you'll need to manually enumerate what's in that parameter (because it might be more than one thing). If input comes from the pipeline, the parameter will only contain one thing at a time, but you can still enumerate it, meaning that you can use the same exact code. This makes the code less complicated, because PowerShell is doing a lot of the hard work under the hood.

The result, shown in listing 21.5, is informally called a *script cmdlet* by the PowerShell community and is formally called an *advanced function* in PowerShell's documentation. Run `help about_functions_advanced*` for help topics.

**Listing 21.5 Making our filtering function into an advanced function**

```
function GetServerInfoWork {
    param([string]$computername,[string]$logfile)
    $os = Get-WmiObject Win32_OperatingSystem -computer $computername

    $disk = Get-WmiObject Win32_LogicalDisk -filter "DeviceID='C:'" `
     -computer $computername

    # use $logfile to get the value from the
    # -logfile parameter

    $obj = New-Object -TypeName PSObject

    $obj | Add-Member -MemberType NoteProperty `
      -Name ComputerName -Value $computername

    $obj | Add-Member -MemberType NoteProperty `
      -Name BuildNumber -Value ($os.BuildNumber)

    $obj | Add-Member -MemberType NoteProperty `
      -Name SPVersion -Value ($os.ServicePackMajorVersion)

    $obj | Add-Member -MemberType NoteProperty `
      -Name SysDriveFree -Value ($disk.free / 1MB -as [int])

    Write-Output $obj
}
function Get-ServerInfo {
    [CmdletBinding()]                                    <-- 1 CmdletBinding directive
    param (
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,              <-- 2 Parameter attributes
                   ValueFromPipelineByPropertyName=$True)]
```

```
        [Alias('host')]
        [string[]]$computername,
        [string]$logfile
    )
    BEGIN {}
    PROCESS {
        foreach ($computer in $computername) {
            GetServerInfoWork -computername $computer `
              -logfile $logfile
        }
}
    END {}
}

#Get-ServerInfo -verbose -computername (Get-Content c:\names.txt)
Get-Content c:\names.txt | Get-ServerInfo | Format-Table -auto
```

**3 Parameter alias**

The changes here were all made to the `Get-ServerInfo` public function. I started by adding the `[CmdletBinding()]` directive **1**, which tells the shell that I'll be using the extended, cmdlet-style parameter attributes. I didn't add any attributes to the `-logfile` parameter, but I added three to the `-computername` parameter, declaring it as mandatory, and indicating that it should accept input from the pipeline both ByValue and ByPropertyName **2**. That means both of these examples will now work:

```
Get-ADComputer -filter * | Select @{l='computername';e={$_.name}} |
Get-ServerInfo

Get-Content names.txt | Get-ServerInfo
```

You'll also see where I declared an alias for the parameter **3**, meaning that the eventual user of this function could use `-host` as well as `-computername`.

In the body of the code, my only change was to remove `$_` and use `$computername` instead. Because the shell now knows that `$computername` is the target for pipeline input, there's no longer any need to use `$_`. When input is piped into the function, `$computername` will contain one object at a time within the `PROCESS` script block, just as `$_` did in the filtering function earlier in this chapter. When input is fed through a parameter, `$computername` will contain all the objects given to the parameter, so I enumerate them using a `ForEach` block.

You can go a bit further with these functions. For example, earlier in this book you learned that the `-confirm` and `-whatif` parameters are supported by most cmdlets that attempt to modify the system. You can add that same kind of support to an advanced function, and PowerShell does most of the work. To illustrate this, let's use a slightly different example, shown in the next listing.

**Listing 21.6 Making a script cmdlet do something dangerous**

```
function RebootWork {
    param([string]$computername)
    Get-WmiObject Win32_OperatingSystem -computer $computername |
     Invoke-WmiMethod -name Reboot | Out-Null
}
```

```
function Reboot-Server {
    [CmdletBinding(SupportsShouldProcess=$True,
                   ConfirmImpact='High')]                    <-- 1 Add SupportsShouldProcess
    param (
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [Alias('host')]
        [string[]]$computername
    )
    BEGIN {}
    PROCESS {
        foreach ($computer in $computername) {
            if ($pscmdlet.ShouldProcess($computer)) {          <-- 2 Test ShouldProcess
                RebootWork -computername $computer
            }
        }
    }
    END {}
}

Get-Content c:\names.txt | Reboot-Server -whatif
```

All I've done is added a bit of language to the cmdlet binding declaration 1, telling it that my function does support the ShouldProcess protocol. That turns on support for the `-whatif` and `-confirm` parameters. In fact, if you were to ask for help on this function, those parameters would be listed even though we haven't added any other help to the function. Simply declaring support isn't sufficient, though: we also need to implement it. That's done whether the computer names came from a parameter or the pipeline, by using the built-in `$pscmdlet` object's `ShouldProcess()` method 2. If the user runs the function with the `-whatif` parameter (as I did on the final line of the script), `ShouldProcess()` will automatically display a what-if message and return `$False`, so that the reboot action doesn't actually happen. If the user runs the function with `-confirm`, the shell will perform the confirmation prompt and return `$True` or `$False` according to the user's confirmation input.

The `ConfirmImpact` declaration 1 has an effect here: the shell has a built-in `$ConfirmPreference` variable that is set to `High` by default. Confirmation prompting happens automatically when `ConfirmImpact` is equal to, or higher than, the `$ConfirmPreference` variable. So, in this function, confirm prompting should happen even if `-confirm` isn't specified. The values for `ConfirmImpact` are `Low`, `Medium`, and `High`, and it's entirely up to your discretion which one you use. I tend to think of these settings as, "what is the likelihood of someone getting fired if they do this accidentally?"

## 21.4 Bundling functions into modules

The next step is to make this function a little easier to distribute, and that involves making it into a *script module*. All that means is saving the file with a specific filename, in a specific location, so that others can load it into the shell by using the `Import-Module` cmdlet. Done properly, it will enable them to use our `Get-ServerInfo` or `Reboot-Server` functions just like any other cmdlet.

Start by removing any code that isn't contained within a function. For listings 21.5 and 21.6, for example, you'd remove the last line or two that I was using to test the function.

You need to come up with a name for the module. I often include multiple different useful functions in a single file and use a name like DonTools. Save your script as *module-name*.psm1, putting in the module name as the main portion of the filename. For example, I'm naming it DonTools.psm1.

With the file saved, you need to decide where to put it, and you have two choices:

- If you don't mind specifying a path to the file, you can put it anywhere. When you use `Import-Module` to load the file, simply provide a full path and filename.
- If you want to be able to load the module without a path (`Import-Module DonTools`), then the module needs to go in a place where PowerShell can find it. By default, the `PSModulePath` environment variable provides one place for your own modules to go (and a second for Microsoft-supplied modules, which we'll ignore because I don't work for them). This is the recommended location.

If you decide to go the second route, here's what you'll need to do:

1. In your Documents folder, create a folder named WindowsPowerShell if one doesn't already exist.
2. In that folder, create a subfolder named Modules.
3. In Modules, create a subfolder with your module's name. In my case, the complete path is /Documents/WindowsPowerShell/Modules/DonTools.
4. Move your module's .psm1 file into that location.

If you'd rather use a file server as a central module repository, you can. Just modify the `PSModulePath` environment variable to include that additional path.

## 21.5 Keeping support functions private

If you saved either listing 21.5 or 21.6 as a module and imported it, take a look at the shell's FUNCTION: drive by running `dir function:`. You'll notice that your worker functions show up, which isn't what you want. Ideally, users should only see the public function, and the worker function should be hidden.

There's an easy way to achieve that: by default, importing a module makes every function inside of it available to users. But if the module includes specific instructions for what should be visible, then only those things will be. Listing 21.7 is a revision of listing 21.6, and you'll see where I added a few specific instructions.

**Listing 21.7 Making a script module that has private functions**

```
function RebootWork {
    param([string]$computername)
    Get-WmiObject Win32_OperatingSystem -computer $computername |
     Invoke-WmiMethod -name Reboot | Out-Null
}
function Reboot-Server {
```

```
    [CmdletBinding(SupportsShouldProcess=$True,
                   ConfirmImpact='High')]
    param (
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [Alias('host')]
        [string[]]$computername
    )
    BEGIN {}
    PROCESS {
        foreach ($computer in $computername) {
            if ($pscmdlet.ShouldProcess($computer)) {
                RebootWork -computername $computer
            }
        }
}
    END {}
}

New-Alias rbt Reboot-Server

Export-ModuleMember -function Reboot-Server
Export-ModuleMember -alias rbt
```

The last three lines of code are the only additions. I've defined an alias, `rbt`, for the `Reboot-Server` function. I've also specified that only that alias and the `Reboot-Server` function should be visible to someone who imports this function into their shell; the `RebootWork` function will remain hidden and inaccessible to users.

## 21.6 Lab

I have two tasks for you to accomplish in this lab. These might take you longer than you have left in the current lunch hour, so feel free to spread them out over a couple of days, if needed.

First, write a filtering function. The ideal use for a filtering function is when you need to pipe in some objects, analyze them in some way, and possibly remove some from the pipeline. In this case, I want your function to accept computer names and ping them. If the computer names are reachable, your function should output them. If they aren't reachable, drop them by simply not outputting them to the pipeline. Call your function something like Test-Host or Ping-Host. You can use the `Test-Connection` cmdlet to perform the ping.

Second, write an advanced function (or script cmdlet, if you prefer that term) that accepts computer names, either from the pipeline or through a `-computerName` parameter. For each computer, display the drive letter and free space (in megabytes) of any local, fixed disk that has less than 10 percent free space. Here's a hint: you'll query the WMI Win32_LogicalDisk class and filter the results so that only those drives having a `DriveType` property of `3` are included. Keep in mind that any given computer might have more than one local disk, so you'll have to account for that and filter the results accordingly. Your worker function should, however, consist entirely of a parameter declaration and a one-line command. There's no need to use an `If`, `ForEach`, or other construct.

## 21.7 Ideas for on your own

We're coming close to the end of this book, with just a few more chapters to go. Hopefully, you're starting to think of some real-world tasks that you'd like to accomplish in PowerShell. Which of those might involve writing a filtering function, an advanced function, or a script module? Start making a list of things you'd like to create, and that list will be a great starting point once you've wrapped up the next few chapters.

# 22 Trapping and handling errors

Anytime you're dealing with computers, errors are bound to occur: network problems, permission denied, server not found ... you know what I'm talking about. Fortunately, your PowerShell commands and scripts can plan for those errors and deal with them, rather than spewing out a bunch of red text.

## 22.1 Dealing with errors you just knew were going to happen

To be clear, I'm not talking about errors that *you* make, such as typos, using the wrong syntax, or something like that. Your errors are called *bugs*, and we'll deal with those in the next chapter. This chapter is going to deal with the errors that are out of your control, but that you can usually anticipate. Here are some examples:

- A "file not found" error
- A "permission denied" error
- The "RPC server not found" error that `Get-WmiObject` can produce
- Other errors related to network connectivity

You can't necessarily prevent these errors from happening, but when they happen you may want to take some specific action. For example, you might want to log the names of computers that can't be reached, or prompt for a different filename if the one specified can't be found. PowerShell offers you a number of ways to deal with these kinds of errors, and we'll cover the two most commonly used ways in this chapter.

## 22.2 Errors and exceptions

First, we need to get some terminology straight. Try running this command:

```
Get-WmiObject Win32_BIOS -computer notonline,localhost
```

Assuming you don't have a computer named NOTONLINE on your network, this command will produce an *error message*, or what I'll refer to as an *error*. It does that because you didn't tell it to do anything else, and because the shell's default action for a nonterminating problem is to display an error and try to keep going. *Nonterminating* simply means that, although the problem interrupted this particular operation, the command is able to continue executing. In this example, it can continue trying the next computer name that was specified.

PowerShell doesn't give you a way to deal with errors like this. It shouldn't; after all, the whole point of its default behavior is to report the problem and keep going. In many cases, that's perfectly acceptable. When you're running a command from the command line, for example, error messages tell you what went wrong, and that's sufficient.

But if you're running a script—especially a script that might be scheduled to run unattended—you won't be around to see the error message, and you'll want the option to do something about the problem, like logging it to a file. To do that, you need to turn the *error* into an *exception*. Another way of saying this is that you need to turn the nonterminating problem into a terminating one, forcing the shell to stop executing the command, and to instead do what you tell it.

## 22.3 The $ErrorActionPreference variable

The shell's default error-handling behavior is defined by a built-in variable called `$ErrorActionPreference`. When you open a new shell session, this variable is set to `Continue`. Its possible values, and their functions, are as follows:

- `SilentlyContinue`—For nonterminating problems, don't display an error message—just keep going.
- `Continue`—For nonterminating problems, display an error message and keep going.
- `Inquire`—For nonterminating problems, ask what to do using an interactive prompt to which the user must respond.
- `Stop`—Stop executing and throw an exception.

Anytime a command runs into a terminating problem from which it can't recover and continue, the behavior is always `Stop`. The exception thrown by `Stop` is something you can *trap* and *handle*.

Please, please, please, please, don't ever put this at the top of a script:

```
$ErrorActionPreference = 'SilentlyContinue'
```

People do that (you'll see it in internet examples all the time) because they anticipate their script having a problem, and they know it's safe to ignore it, and they don't want to see an error message. This is an incredibly poor practice, because it also suppresses any error messages that might help you debug the script. For example, if you edit the script and make a typo somewhere, you won't see an error message when you run the script, because you're suppressing *all* error messages. Instead, if you want to suppress error messages from a particular cmdlet, it's best to do so *just for that cmdlet*, not for the entire script.

## 22.4 The -ErrorAction parameter

The `-ErrorAction` parameter, or its alias `-EA`, is one of the common parameters supported by every cmdlet that runs in PowerShell. Using this parameter, you can override the `$ErrorActionPreference` setting for just that cmdlet.

For example, if you wanted to suppress errors from `Get-WmiObject`, you could do this:

```
Get-WmiObject Win32_Service -computer localhost,notonline
➥ -ea 'SilentlyContinue'
```

**TRY IT NOW** It's safe to run all of the commands and scripts I'll be showing you in this chapter, so please follow along. If you happen to have a computer named NOTONLINE on your network, just substitute something else (NOTHERE, NOTHING, and so on) for NOTONLINE in my examples.

The upside of this technique is that you won't be suppressing all errors in an entire script; you'll only be suppressing the errors that you know for a fact you can safely ignore.

This parameter is also the key to trapping and handling errors. If you don't want to ignore an error, you can set the error behavior to `Stop` for a specific cmdlet. Then, any nonterminating errors encountered by that cmdlet will be turned into terminating exceptions, which you can trap and handle.

A trick with this is that you don't want to have your cmdlet doing more than one thing at a time. That way, if it encounters an error and turns it into a terminating exception, you won't have any work going undone. Accomplishing that trick is easy within a pipeline function. For example, you don't want to do this:

```
Get-WmiObject Win32_BIOS -comp Server-R2,NotOnline,LocalHost -ea Stop
```

Assuming the first and third computers are available, the command will stop running when it fails to reach NOTONLINE. The command will never even try LOCALHOST, and there's no way to trap the error and tell the shell, "OK, go back and finish executing that last command." Instead, we'll build our command so that it only needs to execute against one computer at a time.

This listing shows a brief example, using a pipeline (or filtering) function.

**Listing 22.1 Pipeline functions target one computer at a time**

```
function Get-Stuff {
  PROCESS {
    Get-WmiObject Win32_BIOS -comp $_ -ea Stop
  }
}

'Server-R2','Notonline','Localhost' | Get-Stuff
```

The last line of that script pipes three computer names into the function, and the `PROCESS` script block runs once for each computer name that was piped in. Each time `PROCESS` executes, it places the next computer name into the `$_` placeholder.

Of course, because we aren't handling the error, the end result of this example is the same: once the command hits NOTONLINE, everything stops working. But now we've created a framework in which we can trap and handle the error.

## 22.5 Using a Trap construct

The first technique I'll show you is the more complicated of the two we'll cover. It's called a `Trap` construct. You define this before you anticipate the error occurring, meaning that the construct needs to appear in your script before the command that might generate the error you want to trap. It's possible to declare multiple `Trap` constructs, with each one trapping a different kind of error. In this chapter, we'll stick with a single, generic `Trap` that will catch anything; run `help about_trap` in the shell to see examples of error-specific traps.

The next listing extends listing 22.1, adding a `Trap` construct.

**Listing 22.2 Adding a `Trap` to our pipeline function**

```
function Get-Stuff {
  PROCESS {
    trap {

        $_ | Out-File c:\errors.txt -append
        continue
    }
    Get-WmiObject Win32_BIOS -comp $_ -ea Stop
  }
}

'Server-R2','Notonline','Localhost' | Get-Stuff
```

1 When an exception occurs, the trap executes

2 The command executes first

3 Execution resumes inside the same block

Here's what happens:

1. The last line in the script is the first to execute, and it pipes three computer names into the function.
2. The function executes, running the command 2. On the second execution, the command has a computer name, NOTONLINE, that doesn't exist. It generates a nonterminating error, which `-EA Stop` turns into a terminating exception.
3. When it sees the terminating exception, PowerShell executes the `Trap` construct 1. In it, we pipe the current computer name, which is still in the `$_` placeholder, into a file, appending it to whatever is already in the file.
4. The `Trap` construct ends with the keyword `continue`, which tells the shell to resume execution within the same scope 3. That means the shell stays within the `PROCESS` script block and attempts to execute `Get-WmiObject` with the remaining computer name.

The other way to end a Trap is by using the keyword break. That exits the current scope and passes the exception up to the parent scope. We'll cover scope in just a moment, but before we do, I'd like to offer one minor improvement to the script.

Before the function starts executing its PROCESS script block, it will look for a BEGIN script block and execute that first. We can use that to delete the error log file, so that we get a fresh file each time. It's possible that attempting to do so will generate an error if the file doesn't already exist, so we can suppress that error using -EA. This listing shows the finished function.

**Listing 22.3 Deleting the error file at the beginning of the function**

```
function Get-Stuff {
  BEGIN {
    del c:\errors.txt -ea SilentlyContinue
  }
  PROCESS {
    trap {
        $_ | Out-File c:\errors.txt -append
        continue
    }
    Get-WmiObject Win32_BIOS -comp $_ -ea Stop
  }
}

'Server-R2','Notonline','Localhost' | Get-Stuff
```

## 22.6 Trap scope

Trap constructs are especially sensitive to *scope*, which is a system of containers that the shell applies around certain elements (you learned about them first in chapter 17). When you start using a new PowerShell session, you're in the top-level, *global scope*. When you run a script, the shell creates a new scope around the script, so that anything that happens in the script stays more self-contained. If a script creates a new alias, or a new variable, then all that happens within the script's scope. When the script finishes, its scope is discarded, along with anything that was created within that scope. Functions also get their own scope, as do Trap constructs.

When a terminating exception occurs, the shell looks for a Trap construct within the same scope as whatever command caused the exception. If the shell doesn't find a trap there, it exits that scope and passes the exception up to the parent scope.

This listing illustrates this—an example will be a lot clearer than a long explanation.

**Listing 22.4 Moving the Trap construct to outside the function**

```
trap {
  $_ | Out-File c:\errors.txt -append      <- Trap is within
  continue                                 1 script scope
}

function Get-Stuff {                                    2 Delete file
  BEGIN {
    del c:\errors.txt -ea SilentlyContinue         <-
```

```
  }
  PROCESS {
    Get-WmiObject Win32_BIOS -comp $_ -ea Stop        ◁ 3 Exception generated
  }
}
                                                      4 Command executes first
'Server-R2','Notonline','Localhost' | Get-Stuff       ◁
Write-Host "Done"                                  ◁ 5 Next line in current scope
```

All I've done here is move the `Trap` construct to the beginning of the script, outside the function. Here's what happens when I run this:

1. The command executes first 4, piping three computer names to the function. Right now, we're in the script's scope.
2. The `BEGIN` script block executes, deleting any existing file 2. This enters the function's scope, and the function's scope becomes a child of the script, because the function is contained within the script.
3. The `PROCESS` script block begins executing 3. On its second iteration, it generates a terminating exception because NOTONLINE isn't available, and because `-EA Stop` is the error action.
4. The shell doesn't find a `Trap` construct within the function, so it exits the function's scope, passing the exception. Because the function is a child of the script, the shell exits into the script's scope.
5. Now the shell tries to find a `Trap` within the script's scope, and it finds one 1. It executes the `Trap`, which ends with `continue`.
6. Because we used `continue`, the shell will execute the next line of code in the same scope. Right now, we're in the *script* scope—remember that we *exited* the function's scope to look for a `Trap`. We can't re-enter the function at this point. So the next line of code advises us that the script has finished running 5.

All of this scope stuff can be very hard to keep track of. The easy rule to remember is to put your trap as close as possible to whatever command might generate an error. Doing so keeps you in the same scope as that command, so that you can try to resume execution.

Overall, I find the `Trap` construct to be confusing and hard to keep track of, so I don't use it much. I tend to use it only when I want to have a very top-level way of catching any otherwise-unhandled errors. For more specific errors, PowerShell v2 introduced a `Try` construct, which I find to be easier.

## 22.7 Using a Try construct

The `Try` construct eliminates the need to keep track of scopes, continuing, and so forth. It's a much simpler construct. Like `Trap`, it can be used to provide different actions for different exceptions.

For these examples, I'll be sticking with the simplest form, which handles all exceptions the same way. If you want to see examples of a `Try` construct that handles different exceptions differently, run `help about_try_catch` in the shell.

The following listing shows the revised function, now using a `Try` instead of a `Trap`.

**Listing 22.5 Using a `Try` construct instead of the more complicated `Trap`**

```
function Get-Stuff {
  BEGIN {
    del c:\errors.txt -ea SilentlyContinue
  }
  PROCESS {
    Try {
      Get-WmiObject Win32_BIOS -comp $_ -ea Stop
    } Catch {
      $_ | Out-File c:\errors.txt -append
    }
  }
}

'Server-R2','Notonline','Localhost' | Get-Stuff
```

The differences between listings 22.4 and 22.5 are all within the `PROCESS` script block. I start with the keyword `Try`, and like all constructs, it can contain one or more commands within curly braces. In those braces, I've placed the command that I think might cause a problem. I'm still using `-EA Stop` to ensure that any nonterminating errors become terminating exceptions.

If an exception does occur, the shell will immediately jump to the `Catch` portion of the construct, and execute whatever commands are contained within its curly braces. There's no jumping out of scope like there was with `Trap`, and the shell will always resume execution immediately following the `Catch` portion. In this case, that brings it to the end of the `PROCESS` script block, so it can loop back up and repeat the `PROCESS` script block for its third iteration.

> **TRY IT NOW** Just a quick reminder that you should be typing these in, or downloading them from MoreLunches.com, and reviewing the results. Because some of these commands generate lengthy results, and because the results aren't as important as what's happening in the script, I'll continue to omit the script output and focus on the script itself.

There's another option for using `Try`, which is to add a `Finally` portion. This portion executes whether there is an error or not. Here's a quick example:

```
Try {
  Get-WmiObject Win32_BIOS -comp $_ -ea Stop
} Catch {
  $_ | Out-File c:\errors.txt -append
} Finally {
  Write-Host "Command executed"
}
```

It's legal to have just two of these parts: you *always* have to have the `Try` portion, and you *can* have either a `Catch`, a `Finally`, or both a `Catch` and a `Finally`. As I mentioned earlier, you can also include multiple Catch blocks if you want to handle different exceptions in different ways.

I also mentioned earlier that I like to use Trap constructs as a high-level catch for unanticipated errors, and stick with Try for specific errors on specific commands. That implies that you can use both Trap and Try together, and you can. Here is a quick example of that.

**Listing 22.6 Using Trap and Try in the same script**

```
trap {
  "Unexpected error!" | Out-File c:\errors.txt -append
  continue
}

function Get-Stuff {
  BEGIN {
    del c:\errors.txt -ea SilentlyContinue
  }
  PROCESS {
    Try {
      Get-WmiObject Win32_BIOS -comp $_ -ea Stop
    } Catch {
      $_ | Out-File c:\errors.txt -append
    }
  }
}

'Server-R2','Notonline','Localhost' | Get-Stuff
```

Any errors in Get-WmiObject will be handled by the Try construct. Its Catch portion logs the failed computer name, which is a very specific action. In the event that some terminating exception occurs elsewhere, the shell will eventually find the Trap construct at the top of the script (even if it has to exit the function's scope to do so), and that will log a more generic error message.

Right now, I think the only way the Trap would execute is if Get-WmiObject ran into a problem, and Out-File wasn't able to write to Errors.txt (perhaps because the file was marked as read only). In that case, the shell would have to execute the Trap construct—but it would fail also, because it's trying to write to the same file! You might want to modify this script yourself to handle the top-level error in a different fashion.

## 22.8 The -ErrorVariable parameter

One thing I haven't done so far is look at the error that occurred, and there are plenty of reasons why you might want to do so. For example, logging the actual error message, as opposed to just a computer name, might provide information that lets you better troubleshoot and solve the problem.

There are two ways to access information about an exception. One is to use the built-in $error variable. That variable is a collection, and the first item in the collection ($error[0]) will be the exception that occurred most recently.

I prefer to use the other way, which is to specify a variable that I want an error placed into. That's done by using the `-ErrorVariable` parameter, or its alias `-EV`. This is another one of the common parameters—the same set of parameters that `-EA` came from. All cmdlets support both `-EA` and `-EV`, although cmdlets' help files only list the common parameters set name.

The next listing is another revision to our script, this time specifying that errors be captured in the `$WmiError` variable.

**Listing 22.7 Capturing the error into a variable and logging it to a file**

```
function Get-Stuff {
  BEGIN {
    del c:\retry.txt -ea SilentlyContinue
    del c:\errors.txt -ea SilentlyContinue
  }
  PROCESS {
    Try {
      Get-WmiObject Win32_BIOS -comp $_ -ea Stop -ev WmiError
    } Catch {
      $_ | Out-File c:\retry.txt -append
      $WmiError | Out-File c:\errors.txt -append
    }
  }
}

'Server-R2','Notonline','Localhost' | Get-Stuff
```

I only made a few minor changes to the script:

- In the `BEGIN` script block, I'm now deleting two files. I want to keep the computer names in a separate file, which I'm now calling Retry.txt, so that I can quickly retry those computer names later. I'll log error information to Errors.txt.
- In the `Try` construct, I've added `-EV WmiError`, specifying that any errors be placed into the `$WmiError` variable. Note that the variable name isn't preceded with a `$` symbol here!
- In the `Catch` block, I've changed the filename that the computer name is written to. I'm writing the error to the Errors.txt file. Because I want to access the contents of `$WmiError`, I do use the `$` symbol here.

In case you're curious, here are the contents of Errors.txt after running this script:

```
PS C:\Users\Administrator> gc c:\errors.txt
Get-WmiObject : The RPC server is unavailable. (Exception from HRESULT:
     0x800706BA)
At line:8 char:20
+       Get-WmiObject <<<<  Win32_BIOS -comp $_ -ea Stop -ev WmiError
    + CategoryInfo          : InvalidOperation: (:) [Get-WmiObject],
     COMException
    + FullyQualifiedErrorId :
     GetWMICOMException,Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

```
Command execution stopped because the preference variable
     "ErrorActionPreference" or common parameter is set to
 Stop: The RPC server is unavailable. (Exception from HRESULT: 0x800706BA)
```

## 22.9 Common points of confusion

As I've pointed out, I personally find the `Trap` construct to be a bit confusing and hard to follow. The main reason it's in PowerShell is because it was easier for PowerShell's programmers to create, and it was all they had time to do for v1. Now that v2 has the more sensible `Try` construct, I think that's what most people tend to use.

I want to quickly refocus on this business of when to use the `$` with a variable, and when not to. Technically, the `$` isn't a part of the variable's name. The `$` tells the shell that the following characters will be a variable name, up to the next white space. If the variable name is enclosed in curly braces, the name may contain spaces. When I use the `-EV` parameter, I want to tell it the *name* of the variable I want it to use, and that doesn't include the `$`. When I want to access the *contents* of a variable, I prefix the name with `$`.

For example, suppose I specified `-EV $WmiError`. In that case, the error would be placed into a variable named after *whatever was inside* `$WmiError`! If `$WmiError` was empty at the time, the shell wouldn't know what to do (and would generate another error).

This can be a bit confusing, but it's something you'll have to keep track of: when the shell needs a variable name from you, that name never includes the `$`. When you want to get to the information that's inside a variable, you specify the `$` before the variable name.

## 22.10 Lab

The scripting in listing 22.8 includes several commands that might cause an error, either because a file doesn't exist or because a computer can't be contacted. Add error handling to the script so that it ignores "file not found" errors, and so that it logs the names of any computers that can't be contacted. You can use either kind of error handling, but I suggest sticking with the `Try` construct.

Note that this is slightly tricky because there are two calls to WMI involved. Here's a hint: you can safely assume that if the first WMI command works, the second one will also work.

**Listing 22.8 A script for you to add error handling to**

```
function Get-Inventory {
  BEGIN {
    Remove-Item c:\retries.txt
  }
  PROCESS {
    $os = Get-WmiObject Win32_OperatingSystem -comp $_
    $bios = Get-WmiObject Win32_BIOS -comp $_
    $obj = New-Object -TypeName PSObject
    $obj | Add-Member -MemberType NoteProperty -Name ComputerName
```

```
    ➥ -Value ($_)
    $obj | Add-Member -MemberType NoteProperty -Name OSBuild
    ➥ -Value ($os.buildnumber)
    $obj | Add-Member -MemberType NoteProperty -Name SPVersion
    ➥ -Value ($os.servicepackmajorversion)
    $obj | Add-Member -MemberType NoteProperty -Name BIOSSerial
    ➥ -Value ($bios.serialnumber)
    Write-Output $obj
  }
}

'localhost','server-r2' | Out-File c:\names.txt
Get-Content names.txt | Get-Inventory | Export-CSV c:\inventory.csv
```

## 22.11 Ideas for on your own

Make a short list of other errors that you might anticipate, and the commands that might cause them. As you start writing your own scripts in the future, refer to that list. When you find yourself using a command that you think might generate an error, build in your error-handling right from the start.

# 23 Debugging techniques

Anytime you've typed more than two letters on your keyboard, you have created an opportunity for mistakes to creep in. In the programming world, those mistakes are called *bugs*. Although using PowerShell isn't necessarily programming, we PowerShell jockeys use the word *bug*, too.

Fun story: The word *bug*, as applied to computers behaving incorrectly, actually came from a real insect (a moth) that got trapped inside Harvard University's Mark II Aiken Relay Calculator. The moth got stuck in one of the computer's relays, causing the computer to generate incorrect results. The whole story is told at http://www.jamesshuggins.com/h/tek1/first_computer_bug.htm.

## 23.1 An easy guide to eliminating bugs

There are two broad categories of bugs in the PowerShell world (actually, this applies to software in general, but we'll stick with PowerShell). The first category is *syntax errors*, and the second is *logic errors*.

### 23.1.1 Syntax errors

Syntax errors are by far the easiest to deal with, and we won't spend much time on them in this chapter. A syntax error simply means you typed something wrong. It might be a command name that you misspelled (`Gte-Content` instead of `Get-Content`, for example), or it might be that you got the actual syntax of the command wrong (forgetting the hyphen between the verb and noun of a cmdlet name, or using a colon instead of a space to separate a parameter's name and value). Whatever the cause, correcting your typing will solve the problem.

Best of all, PowerShell will usually tell you, in explicit detail, where the problem is. PowerShell might not always know *what* the problem is, but it will usually get pretty close to the *location* of the error. For example, here's a syntactically incorrect command and the resulting error message:

```
PS C:\> get-content -file names.txt
Get-Content : A parameter cannot be found that matches parameter name
 'file'.
At line:1 char:18
```

That's pretty clear: I used a parameter named `-file`, and PowerShell couldn't find one. The error occurred on line 1 of my command, at character position 18. You might notice that character position 18 is where the parameter value, names.txt, is located. It's a bit odd that PowerShell chose to pinpoint that as the location of the error, but once you understand that that's how it works, you'll understand future error messages.

Ultimately, the problem here is that *I didn't read the help file* to find out what parameters were available. Fixing the problem is as simple as reading that help file, and seeing that `-path` is the parameter I'm after, not `-file`.

You can help yourself avoid this kind of error by using a quality third-party PowerShell console or editor, such as SAPIEN PrimalScript (www.primalscript.com), Idera PowerShell Plus (www.idera.com), or PowerGUI (www.powergui.org; there are both free and commercial versions). These products all include a few common features:

- *Code hinting*—Reminds you of the available parameters and helps type them for you—saving time, and helping to protect against typos.
- *Syntax highlighting*—Colors valid syntax elements in a specific way, with invalid syntax often getting a different color. That helps to visually alert you to a potential problem.
- *Live syntax checking*—Works a bit like the spell-check feature in Microsoft Word: the product puts a red underline underneath bits it doesn't think are correct, such as invalid parameter names.

You might not believe this, but I regularly see students struggling with simple syntax errors, mostly because they won't take the time to read the actual error message. I understand where they're coming from. Frankly, I freak out a little bit when all that red text starts spilling across the screen. I think it reminds me of grade school, when my teachers were ruthless with that red pen. But if you slow down, read the error message, and think about what it's saying, you can usually point your eyes directly at the problem. Read the help, and look at some of the examples in the help to see if you can figure out the correct way to proceed. Double-check your punctuation in particular.

Here's a checklist:

- Make sure you typed the cmdlet name correctly. If you used an alias, make sure it's typed correctly, and that it points to the cmdlet you think it does. You can run `Get-Alias alias` (insert your alias name for `alias`) to double-check which cmdlet an alias points to.

- Make sure parameter names are preceded by a dash and are followed by a space. Make sure you're using the correct parameter name (read the help!), and if you're abbreviating the parameter name, make sure you're providing enough characters to uniquely identify that parameter.
- Most of PowerShell's punctuation comes in pairs: single quotes, double quotes, square brackets, curly braces, and parentheses are all good examples. Make sure that you end every set that you start, and that you properly nest them. Improper nesting, like `({this)}`, means you're ending a pair before ending the pair it encloses. In that example, I closed the parentheses before the curly braces, which is the opposite of the correct order.
- Watch your spaces. In PowerShell, spaces are special characters that indicate a separation between command elements. PowerShell isn't that case-sensitive (meaning that upper- and lowercase are usually the same to the shell), but it's very space-sensitive. There's a space after a cmdlet name and before any parameters or values. There's a space in between parameter names and values. There's a space after one parameter and before the next. Don't forget those.

**Above and beyond**

I'm not kidding about the red text freaking me out. It's especially embarrassing when someone is looking over my shoulder. Half the time, I'll just run `Cls` (which is an alias to `Clear-Host`) to make it all go away, rather than reading the message. Try getting an error in the middle of a conference session demonstration, with a thousand people watching you! No pressure!

So here's a trick I use: I'll change the color of the error message text to green. Seriously, I do.

```
(Get-Host).PrivateData.ErrorForegroundColor = 'green'
```

That only lasts for the duration of the shell session, so I'll either put that in a profile script (more on those in chapter 24), or I'll just remember to do it before I start working in the shell. The green text makes me feel a lot better, and it doesn't look so aggressive when it pops up in the middle of a demonstration.

You can also change the `ErrorBackgroundColor`, `WarningForegroundColor`, `WarningBackgroundColor`, and other colors. TechNet has a nice "Modifying Message Colors" article on the available options (http://mng.bz/1037). I'll also cover these in more detail in chapter 24.

That's really everything you need to know about syntax errors. They're a pain in the neck, but they shouldn't be that difficult to fix. Just pay close attention to what you're typing.

### 23.1.2 Logic errors

Logic errors mean that your script or command isn't doing what you want it to do, but it's not necessarily generating an error.

Some logic errors will produce straightforward errors. You should know what to do with a "file not found" error, for example, or an "access denied" message, but sometimes errors aren't always so clear. `Get-WmiObject`, for example, can produce an "RPC server not found" error if it's not able to locate a remote computer, or if that computer can't accept the WMI connection (perhaps because of a firewall or a permissions issue).

But the most vexing logic errors are the ones that don't produce any error at all—they just prevent your script or command from working properly. This next listing is an example—go ahead and open this script in the PowerShell ISE and run it, or just run it from the PowerShell console host.

**Listing 23.1 A short script containing logic errors**

```
$name = Read-Host "Enter computer name"
if (test-connection $name) {
    get-wmiobject win32_bios -computername $nmae
}
```

Logic errors, like syntax errors, can come from typos, and one of the errors in listing 23.1 is a simple typo. Logic errors also come from what I call a *bad assumption*: you're assuming that a particular variable, cmdlet output, or property contains one thing, when, in fact, it contains something entirely different. Although listing 23.1 is pretty short, it manages to contain a bad assumption as well as a typo.

Debugging causes a lot of frustration for a lot of administrators. I'll try and make it simpler by telling you exactly what you need to know to debug any script or command, no matter how complicated it is:

- You can't debug a script or command unless you have a clear expectation of what it's going to do.
- You must execute your script and examine its reality (what it actually does), and compare that reality to your expectations. When reality and your expectations differ, you have found the bug.
- While executing the script and examining it, you need to read very, very carefully, so that you can spot typos. Sometimes using a different font can help.

In the next three sections, I'll use listing 23.1 as an example, and show you different ways to debug it.

## 23.2 Identifying your expectations

If you don't know what a command or script should do, then you can never debug it. Period, end of statement, thanks for reading. That's why my first step in debugging is to document my exact expectations. I'm experienced enough that I can often do so in my head for a short script or simpler command, but for more complex ones I dig out a piece of paper and a pen and write everything down. We're doing to do that with listing 23.1, so that you can see the process involved. I can't tell you enough how useful this approach has been in helping me debug some incredibly complicated scripts, and I encourage you to use this approach whenever you have to debug anything.

I want to emphasize that I'm documenting my expectations; that isn't necessarily going to be the same as what is correct. I'm not *teaching* you how listing 23.1 works at this point. Instead, I'm telling you what I expect it to do, based on a quick read-through of the commands.

Let's begin with line 1:

```
$name = Read-Host "Enter computer name"
```

I expect that this will display "Enter computer name:" on the screen, and allow me to type something. Whatever I type will be stored in the variable `$name`.

Line 2:

```
if (test-connection $name) {
```

My expectation is that this will run the `Test-Connection` command, passing it the computer name in `$name`. In other words, I expect that this will ping the computer name I typed. I see that this is enclosed in an `If` construct, so I expect that `Test-Connection` must return a `True` or `False` value. So, if the computer can be pinged, it will return `True`, and whatever is inside the `If` construct will execute next.

Line 3:

```
get-wmiobject win32_bios -computername $nmae
```

I expect that this will run `Get-WmiObject` and retrieve the Win32_BIOS class. I've used `Get-WmiObject` before, and I know that the `-class` parameter is in the first position, so Win32_BIOS is going to be fed to the `-class` parameter. I see that the `-computerName` parameter is also specified, and it's being passed the computer name from the `$name` variable. Oh, wait—there's a typo. See, just a careful read-through of the script found a problem. I'm going to leave the typo in there, though, and pretend that I wasn't being so careful with it. I want to show you some other ways that you might have found it.

Finally, line 4:

```
}
```

That closes the `If` construct. Were I using one of the third-party editors that I mentioned earlier, I might even use a brace-matching feature to double-check the `If` construct's braces. Such a feature will highlight all the code in between two braces (or brackets, or parentheses, or quotes, or whatever), so that you can visually verify that you ended everything you started. If you're using one of those products, consult its documentation on how to check brace or construct matching.

With my expectations written down, it's time to start seeing where they differ from reality.

## 23.3 Adding trace code

The first trick I'll show you is to add trace code to the script. It all starts with a helpful command called `Write-Debug`, which simply takes a message that you want it to display:

```
Write-Debug "Test message"
```

**TRY IT NOW** See if you can run this command in PowerShell.

If you're following along, you'll notice that `Write-Debug` doesn't produce any output. Not very useful, is it?

`Write-Debug` actually sends your message to an *alternate pipeline*, called the Debug pipeline. PowerShell has several of these alternate pipelines: Error, Warning, Debug, and so forth. Each of them is controlled by a kind of on/off switch called a *preference variable*. By default, the Debug pipeline's switch is set to `SilentlyContinue`, which is the same as `Off`. The result is that all `Write-Debug` messages are suppressed, or hidden, by default.

To see the debug messages, you need to change the value of that on/off switch, and it can be changed in several places. If you change it in the shell itself, the change will affect everything that happens in that shell session, which isn't necessarily what you want. Alternately, if you change the setting from within a script, it'll only affect that script—everything else you do in the shell will be unchanged.

For this example, I'll change the setting just in the script by adding this to the top:

```
$DebugPreference = "Continue"
```

Now I'm free to add `Write-Debug` statements to my script. The next listing shows the revised script.

**Listing 23.2 Script with trace output added**

```
$DebugPreference = "Continue"

$name = Read-Host "Enter computer name"
write-debug "`$name contains $name"

if (test-connection $name) {
    write-debug "Test-connection was True"
    get-wmiobject win32_bios -computername $nmae
} else {
    write-debug "Test-connection was False"
}
```

The first addition was a call to `Write-Debug`, asking it to display the contents of the variable `$name`. What I've done here is a really cool trick:

- Because I used double quotation marks, PowerShell will look for the `$` character. Whenever it sees it, the shell will assume that all following characters, to the next white space, form a variable name. The shell will then replace the variable name, and the `$` character, with the contents of that variable.
- The first time I refer to the variable, I precede the `$` character with the backtick (`` ` ``) character, which is PowerShell's escape character. That character can be really tough to distinguish from a single quote in certain fonts, but trust me, it's different. It's on the upper left of a U.S. keyboard, on the same key as the tilde (~) character. The backtick takes away the special meaning of the `$` character, so that PowerShell doesn't "see" the first `$name` as a variable.

**TRY IT NOW** Try running this script in the shell, and see what happens.

You'll also notice that I added a `Write-Debug` to the inside of the `If` construct. I even added an `Else` portion to the construct, containing a third `Write-Debug` message. That way, no matter which way the `If` construct's logic goes, I'll see some output and know what's happening inside the script.

If you're following along, you should have seen the following output (assuming you entered SERVER-R2 for the computer name):

```
Enter computer name: SERVER-R2
DEBUG: $name contains SERVER-R2
DEBUG: Test-connection was True
Get-WmiObject : Cannot validate argument on parameter 'ComputerName'.
 The argument is null or empty. Supply an argument that is not null o
r empty and then try the command again.
At C:\demo.ps1:8 char:43
```

That's where you'll realize that there's a typo in that `$name` variable. PowerShell is clearly telling us that there's a problem with the `-computerName` parameter; if we look carefully at just that portion of the script, the `$nmae` typo is more obvious.

I'll fix that now and run the script again. Here's a portion of the output:

```
Enter computer name: SERVER-R2
DEBUG: $name contains SERVER-R2
DEBUG: Test-connection was True


SMBIOSBIOSVersion : 6.00
Manufacturer      : Phoenix Technologies LTD
Name              : PhoenixBIOS 4.0 Release 6.0
```

That looks like what I want.

Now I need to test the opposite situation: what happens when I provide a computer name that isn't valid?

```
Enter computer name: nothing
DEBUG: $name contains nothing
Test-Connection : Testing connection to computer 'nothing' failed: Th
e requested name is valid, but no data of the requested type was foun
d
At C:\demo.ps1:6 char:20
```

Oops. Not what I was hoping for. That's a logic error: the `Test-Connection` cmdlet clearly isn't doing what I expected, which was to return a simple `True` or `False` value.

Let's step out of the script, and just work with the `Test-Connection` cmdlet from the command line:

```
PS C:\> test-connection server-r2

Source        Destination     IPV4Address      IPV6Address
------        -----------     -----------      -----------
SERVER-R2     server-r2       192.168.10.10    fe80::ec31:bd61:d42...
SERVER-R2     server-r2       192.168.10.10    fe80::ec31:bd61:d42...
SERVER-R2     server-r2       192.168.10.10    fe80::ec31:bd61:d42...
SERVER-R2     server-r2       192.168.10.10    fe80::ec31:bd61:d42...
```

```
PS C:\> test-connection nothing
Test-Connection : Testing connection to computer 'nothing' failed: Th
e requested name is valid, but no data of the requested type was foun
d
At line:1 char:16
```

Okay, that's definitely not what I expected. When I use the command with a valid computer name, I get back a table of results, not a `True` or `False` value. When I use it with an invalid computer name, I still get an error.

Time to read the help, by running `Help Test-Connection -full`. The `-full` part is very important, because I want very detailed information on the cmdlet and its behavior. Reading through the help, I see that the command "returns the echo response replies." The help also says that, "unlike the traditional 'ping' command, `Test-Connection` returns a `Win32_PingStatus` object … but you can use the `-quiet` parameter to force it to return only a Boolean value." Yes, please!

Looking at the `-quiet` parameter, I see it "suppresses all errors and returns `$True` if any pings succeed and `$False` if all failed." That's what I want, so I'll modify the script accordingly:

```
$DebugPreference = "Continue"

$name = Read-Host "Enter computer name"
write-debug "`$name contains $name"

if (test-connection $name -quiet) {
    write-debug "Test-connection was True"
    get-wmiobject win32_bios -computername $name
} else {
    write-debug "Test-connection was False"
}
```

And I'll run it again, testing it with both a valid and an invalid computer name. Here's the output:

```
PS C:\> ./demo
Enter computer name: server-r2
DEBUG: $name contains server-r2
DEBUG: Test-connection was True

SMBIOSBIOSVersion : 6.00
Manufacturer      : Phoenix Technologies LTD
Name              : PhoenixBIOS 4.0 Release 6.0
SerialNumber      : VMware-56 4d 45 fc 13 92 de
                    5b 86
Version           : INTEL  - 6040000

PS C:\> ./demo
Enter computer name: nothing
DEBUG: $name contains nothing
DEBUG: Test-connection was False
```

That's what I want, so the script is now debugged. I don't need to remove all the `Write-Debug` statements, either. In fact, I may want to leave them there in case I ever need to debug this again. For now, I'll just set `$DebugPreference` back to its default of `SilentlyContinue`, which will suppress the output of `Write-Debug`. (Doing that in no way harms the performance of the script, and it's a very convenient way to switch between production mode and debugging mode.)

Here's the script one last time:

```
$DebugPreference = "SilentlyContinue"

$name = Read-Host "Enter computer name"
write-debug "`$name contains $name"

if (test-connection $name -quiet) {
    write-debug "Test-connection was True"
    get-wmiobject win32_bios -computername $name
} else {
    write-debug "Test-connection was False"
}
```

Whenever I begin working on a script, I add the `Write-Debug` statements as I go. I know there will be a bug or two in there eventually, so building in the debugging from the start makes debugging faster when the time comes. Here are my guidelines:

- Whenever I change the contents of a variable, I use `Write-Debug` to output the variable, just so I can check those contents.
- Whenever I'm going to read the value of a property or a variable, I use `Write-Debug` to output that property or variable, so that I can see what's going on inside the script.
- Any time I have a loop or logic construct, I build it in such a way that I get a `Write-Debug` message no matter how the loop or logic works out. In this example, I added an `Else` section specifically to have debug output—the `Else` portion of the construct has no other purpose.

Of course, in a really long script, having to wade through a lot of debug messages to track down a problem can be time-consuming. Is there a more efficient technique? You bet there is!

## 23.4 Working with breakpoints

We're going to revert back to the original script in listing 23.1. It isn't a long script, but it will suffice to illustrate *breakpoints*, a great feature of PowerShell v2. Note that this discussion will focus solely on what's available in the PowerShell console host and the PowerShell ISE; third-party editors often provide similar (and better) breakpoint functionality, but they typically implement it in a different way. You'll have to consult your product's manual if you're using one of those editors.

A *breakpoint* is a defined area where a script will pause its execution, allowing you to examine the environment that the script is running within. PowerShell can be configured to break when

- Your script reaches a certain line
- A variable is read and/or changed
- A specific command is executed

In the first instance, you must specify the script file that you're referring to. In the second and third situations, you can choose to specify a script, and the breakpoint will only be active for that script. If you don't, the breakpoint will occur globally throughout the shell when that variable is read or written, or that command is executed.

Going back to listing 23.1, suppose I want to have the script stop immediately after line 1 finishes executing, meaning that I want to break before line 2. I've saved the script as C:\Demo.ps1, so in the shell I'll run this command:

```
PS C:\> set-psbreakpoint -script c:\demo.ps1 -line 2

  ID Script          Line Command        Variable       Action
  -- ------          ---- -------        --------       ------
   0 demo.ps1           2
```

The shell confirms that it has set the breakpoint. I also want to be notified whenever the `$name` variable is accessed, so I'll run this:

```
PS C:\> set-psbreakpoint -script c:\demo.ps1 -variable name -mode read

  ID Script          Line Command        Variable       Action
  -- ------          ---- -------        --------       ------
   1 demo.ps1                            name
```

Again, the shell confirms. Notice that the variable's name is just `name`, and not `$name`. Variable names don't include the dollar sign; `$` is just a cue to the shell telling it that you wish to use the contents of a variable. In this case, I don't want to refer to the contents of `$name`; I want to refer to the variable `name` itself.

With those two breakpoints set, I'll run the script. After entering the computer name, the script will break. The shell modifies the command-line prompt, indicating that I'm in suspend mode. Here I can examine the contents of variables, execute commands, and so on. When I'm done, I can run `Exit` to resume script execution. Here's how it all looks:

```
PS C:\> ./demo
Entering debug mode. Use h or ? for help.

Hit Line breakpoint on 'C:\demo.ps1:2'

demo.ps1:3   $name = Read-Host "Enter computer name"
[DBG]: PS C:\>>> exit
Enter computer name: server-r2
Hit Variable breakpoint on 'C:\demo.ps1:$name' (Read access)
```

```
demo.ps1:4   write-debug "`$name contains $name"
[DBG]: PS C:\>>> $name
server-r2
[DBG]: PS C:\>>> test-connection $name -quiet
True
[DBG]: PS C:\>>>
```

This gives me the chance to test commands, see what's inside variables or properties, and so forth, without having to add a lot of `Write-Debug` commands. You'll notice, in fact, that the shell generates its own debug output automatically as part of the breakpoint process, and that it automatically turns the Debug pipeline on for the duration of the script's execution.

When I'm done debugging, I can remove the breakpoints:

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

And I can go back to executing my script normally. Working this way takes a bit of getting used to, but it's a very effective debugging tool once you do so.

Breakpoints are also supported within the PowerShell ISE. To set a line breakpoint, move your cursor to the desired line and press F9. You can still set command and variable breakpoints, but you have to run `Set-PSBreakpoint` from the command pane—there's no function key or graphical shortcut. The ISE will visually indicate where line breakpoints occur, using a red highlight. If you run the script within the ISE, breakpoint lines will be highlighted in yellow when the script reaches one of those lines. At that time, you can hover your cursor over any variable to see a tooltip with the contents of that variable. It's a vaguely similar experience to working in a full-fledged development environment like Visual Studio, albeit with much more simplistic functionality.

## 23.5 Common points of confusion

The biggest single mistake I see students make when it comes to debugging is something I call *shotgun debugging*. It means they see an error, they panic, and they start changing everything they can, without taking the time to verify what was wrong.

Take my original example in listing 23.1: I've used that same example in dozens of classes, and as you know, that example is purposely buggy. When I ask the class to try to fix it, there's almost always one student who spends half an hour checking network connectivity, logging off and logging back on, rebooting the remote computer, and so on, assuming all the while that the script must be fine, and that the problem lies in the infrastructure somewhere.

Don't get caught in that trap. When a script or command isn't working the way you think it should, say three things to yourself:

- I need to figure out exactly what I think each line of this script is supposed to do.
- I need to assume that every command is incorrect, and read the help to verify each parameter. I should run each command individually in a test environment to make sure it works the way the script thinks it works.

- I need to examine property and variable contents to make sure they contain what I think they contain.

If you can discipline yourself to slowing down, taking a deep breath, and debugging in a calm, methodical fashion that focuses on expectations versus reality, you'll never meet a script you can't debug.

## 23.6 Lab

Listing 23.3 is a script that includes a function. At the end of the script (on line 14) is a command that actually runs the function. The goal is to get a table that includes each computer's name, its Windows build number, and its BIOS serial number. The script definitely has bugs in it—both syntax and logic. Fix them.

**Listing 23.3 Chapter 23 lab script**

```
function Get-Inventory {
    PROCESS {
        $computername = $_
        $os = Get-WmiObject Win32_OperatingSystem -comp $computername
        $bios = Get-WmiObject Win32_BIOS -comp $computername
        $ojb = New-Object PSObject
        $obj | Add-Membrr NoteProperty ComputerName $computername
        $obj | Add-Member NoteProperty OSBuild ($os.buildnumber)
        $obj | Add-Member NoteProperty BIOSSerial ($bios.serialno)
        Write-Output $ojb
    }
}

localhost,server-r2 | get-inventory
```

Remember that one of my suggested best practices is for you to always spell out full cmdlet and parameter names in a script, and that includes scripts given to you by other people. That would be a good place to start with this one: correcting the names. In the case of the `Add-Member` cmdlet, this script doesn't even include parameter names—they're all positional. You should consider correcting that, too, by adding in the appropriate parameter names. Use the help files to guide you.

As a tip, SAPIEN PrimalScript includes a feature on its edit menu that will convert aliases to their full cmdlet names. You may be able to find a free plug-in for PowerGUI that does something similar, and perhaps also expands the parameter names. If you use a different editor (other than the PowerShell ISE), ask the vendor to supply a cmdlet and parameter name expansion feature in a future version.

# 24 Additional random tips, tricks, and techniques

We're nearing the end of your "month of lunches," and the next chapter is your final exam, where you'll tackle a complete administrative task from scratch. Before you do, I'd like to share a few extra tips and techniques to round out your education.

## 24.1 Profiles, prompts, and colors: customizing the shell

Every PowerShell session starts out the same: the same aliases, the same PSDrives, the same colors, and so forth. Why not make the shell a little bit more customized?

### 24.1.1 PowerShell profiles

I've explained before that there's a difference between a PowerShell hosting application and the PowerShell engine itself. A hosting application, such as the console or the PowerShell ISE, is a way for you to send commands to the actual PowerShell engine. The engine executes your commands, and the hosting application is responsible for displaying the results. Another thing that the hosting application is responsible for doing is loading and running *profile scripts* each time the shell starts.

These profile scripts can be used to customize the PowerShell environment, by loading snap-ins or modules, changing to a different starting directory, defining functions that you'll want to use, and so forth. For example, here's the profile script that I use on my computer.

**Listing 24.1 Don's PowerShell profile script**

```
Import-Module ActiveDirectory
Add-PSSnapin SqlServerCmdletSnapin100
cd c:\
```

My profile loads the two shell extensions that I use the most, and it changes to the root of my C: drive, which is where I like to begin working. You can put any commands you like into your profile.

There's no default profile, and the exact profile script that you create will depend a bit upon how you want it to work. Details are available if you run `help about_profiles`, but you mainly need to consider whether or not you'll be working in multiple hosting applications. For example, I tend to switch back and forth between the regular console and the PowerShell ISE, and I like to have the same profile running for both, so I have to be careful to create the right profile script file in the right location. I also have to be careful about what goes into that profile, because I'm using it for both the console and the ISE—some commands that tweak console-specific settings like colors can cause an error when run in the ISE.

Here are the files that the console host tries to load, and the order in which it tries to load them:

1. $pshome/profile.ps1—This will execute for all users of the computer, no matter which host they're using (remember that `$pshome` is predefined within PowerShell and contains the path of the PowerShell installation folder).
2. $pshome/Microsoft.PowerShell_profile.ps1—This will execute for all users of the computer if they're using the console host. If they're using the PowerShell ISE, the $pshome/Microsoft.PowerShellISE_profile.ps1 script will be executed instead.
3. $home/Documents/WindowsPowerShell/profile.ps1—This will execute only for the current user (because it lives under the user's home directory), no matter which host they're using.
4. $home/Documents/WindowsPowerShell/Microsoft.PowerShell_profile.ps1—This will execute for the current user if they're using the console host. If they're using the PowerShell ISE, the $home/Documents/WindowsPowerShell/Microsoft.PowerShellISE_profile.ps1 script will be executed instead.

If one or more of these scripts doesn't exist, there's no problem. The hosting application will simply skip it and move on to the next one.

On 64-bit systems, there are variations for both 32- and 64-bit scripts, since there are separate 32- and 64-bit versions of PowerShell itself. You won't always want the same commands run in the 64-bit shell as you do the 32-bit shell. For example, some modules and other extensions are only available for one or the other architecture, so you wouldn't want a 32-bit profile trying to load a 64-bit module into the 32-bit shell, because it won't work!

Note that the documentation in `about_profiles` is different from what I've listed here, and my experience is that the preceding list is correct. Here are a few more points about that list:

- `$pshome` is a built-in PowerShell variable that contains the installation folder for PowerShell itself; on most systems, that's in C:\Windows\System32\WindowsPowerShell\v1.0 (for the 64-bit version of the shell on a 64-bit system).
- `$home` is another built-in variable that points to the current user's profile folder (such as C:\Users\Administrator).
- I've used "Documents" to refer to the Documents folder, but on some versions of Windows it will be "My Documents."
- I've written "no matter which host they're using," but that technically isn't true. It's true of hosting applications (the console and the ISE) written by Microsoft, but there's no way to force the authors of non-Microsoft hosting applications to follow these rules.

Because I want the same shell extensions to load whether I'm using the console host or the ISE, I chose to customize $home\Documents\WindowsPowerShell\profile.ps1, because that profile is run for both of the Microsoft-supplied hosting applications.

> **TRY IT NOW** Why don't you try creating one or more profile scripts for yourself? Even if all you put in them is a simple message, such as `Write "It Worked"`, this is a good way to see the different files in action. Remember that you have to close the shell (or ISE) and re-open it to see the profile scripts run.

Keep in mind that profile scripts are scripts and are subject to your shell's current execution policy. If your execution policy is `Restricted`, your profile won't run; if your policy is `AllSigned`, your profile must be signed. Chapter 14 discussed the execution policy and script signing.

### 24.1.2 Customizing the prompt

The PowerShell prompt—the `PS C:\>` that you've seen through much of this book—is generated by a built-in function called `Prompt`. If you want to customize the prompt, you can simply replace that function. Defining a new `Prompt` function is something that can be done in a profile script, so that your change takes effect each time you open the shell.

Here's the default prompt:

```
function prompt
{
    $(if (test-path variable:/PSDebugContext) { '[DBG]: ' }
    else { '' }) + 'PS ' + $(Get-Location) `
    + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}
```

This prompt first tests to see if the `$DebugContext` variable is defined in the shell's VARIABLE: drive. If it is, this function adds `[DBG]:` to the start of the prompt. Otherwise, the prompt is defined as `PS` along with the current location, which is returned by the Get-Location cmdlet. If the shell is in a nested prompt, as defined by the built-in $nestedpromptlevel variable, the prompt will have >> added to it.

This next listing is an alternative prompt function. You could enter this directly into any profile script to make it the standard prompt for your shell sessions.

**Listing 24.2 Custom PowerShell prompt**

```
function prompt {
 $time = (Get-Date).ToShortTimeString()
 "$time [$env:COMPUTERNAME]:> "
}
```

This alternative prompt displays the current time, followed by the current computer name (which will be contained within square brackets). Note that this leverages PowerShell's special behavior with double quotation marks, in which the shell will replace variables (like $time) with their contents.

### 24.1.3 Tweaking colors

In previous chapters, I've mentioned how stressed-out I can get when a long series of error messages scrolls by in the shell. I always struggled in English class when I was a kid, and seeing all that red text reminds me of the essays I'd get back from Ms. Hansen, all marked up with a red pen. Yuck. Fortunately, PowerShell gives you the ability to modify most of the default colors it uses.

The default text foreground and background colors can be modified by clicking on the control box in the upper-left corner of PowerShell's window. From there, select Properties, and then select the Colors tab, which is shown in figure 24.1.

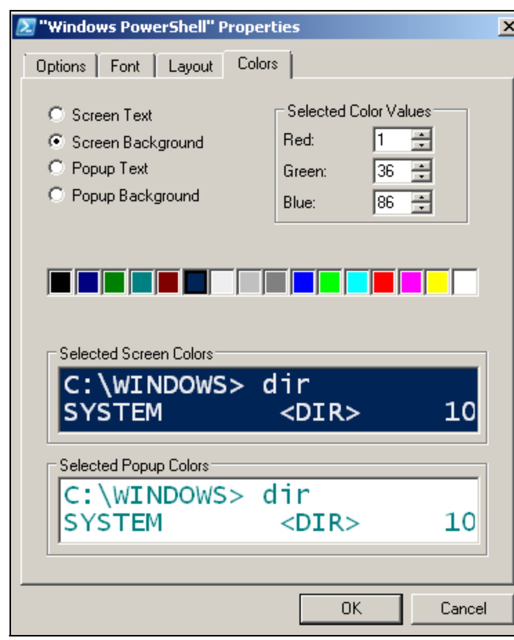



**Figure 24.1 Configuring the default shell screen colors**

Modifying the colors of errors, warnings, and other messages is a bit trickier and requires you to run a command. But you could put this command into your profile, so that it executes each time you open the shell. Here's how to change the error message foreground color to green, which I find a lot more soothing:

```
(Get-Host).PrivateData.ErrorForegroundColor = "green"
```

You can change colors for the following settings:

- ErrorForegroundColor
- ErrorBackgroundColor
- WarningForegroundColor
- WarningBackgroundColor
- DebugForegroundColor
- DebugBackgroundColor
- VerboseForegroundColor
- VerboseBackgroundColor
- ProgressForegroundColor
- ProgressBackgroundColor

And here are some of the colors you can choose:

- Red
- Yellow
- Black
- White
- Green
- Cyan
- Magenta
- Blue

There are also dark versions of most of these colors: DarkRed, DarkYellow, DarkGreen, DarkCyan, DarkBlue, and so on.

## 24.2 Operators: -as, -is, -replace, -join, -split

In chapter 8, I briefly introduced you to the -as operator, and there are four additional operators that I'd like you to know about.

### 24.2.1 -as and -is

The -as operator produces a new object in an attempt to convert an existing object into a different type. For example, if you have a number that contains a decimal (perhaps from the result of a division operation), you can drop the decimal portion by converting, or *casting*, the number to an integer:

```
1000 / 3 -as [int]
```

The object to be converted comes first, then the -as operator, and then, in square brackets, the type you want to convert to. Types can include [string], [xml], [int], [single], [double], [datetime], and others, although those are probably the ones you'll use the most. Technically, this example of converting to an integer will round the fractional number to an integer, rather than just truncating the fractional portion of the number.

The `-is` operator works similarly: it's designed to return `True` or `False` if an object is of a particular type or not. Here are a few one-line examples:

```
123.45 -is [int]
"SERVER-R2" -is [string]
$True -is [bool]
(Get-Date) -is [datetime]
```

**TRY IT NOW** Try running each of these one-line commands in the shell to see the results.

### 24.2.2 -replace

The `-replace` operator is designed to locate all occurrences of one string within another and replace those occurrences with a third string:

```
PS C:\> "192.168.34.12" -replace "34","15"
192.168.15.12
```

The source string comes first, followed by the `-replace` operator. Then you provide the string you want to search for within the source, followed by a comma and the string you want to use in place of the search string. In the preceding example, I replaced "34" with "15."

### 24.2.3 -join and -split

The `-join` and `-split` operators are designed to convert arrays to delimited lists and vice versa.

For example, suppose I created an array with five elements:

```
PS C:\> $array = "one","two","three","four","five"
PS C:\> $array
one
two
three
four
five
```

This works because PowerShell treats a comma-separated list as an array automatically. Now, let's say I want to join this array together into a pipe-delimited string—I can do that with `-join`:

```
PS C:\> $array -join "|"
one|two|three|four|five
```

Saving that result into a variable will let me reuse it, or even pipe it out to a file:

```
PS C:\> $string = $array -join "|"
PS C:\> $string
one|two|three|four|five
PS C:\> $string | out-file data.dat
```

The `-split` operator does the opposite: it takes a delimited string and makes an array from it. For example, suppose you have a tab-delimited file containing one line and four columns. Displaying the contents of the file might look like this:

```
PS C:\> gc computers.tdf
Server1 Windows East    Managed
```

Keep in mind that `Gc` is an alias for `Get-Content`.

You can use the `-split` operator to break that into four individual array elements:

```
PS C:\> $array = (gc computers.tdf) -split "`t"
PS C:\> $array
Server1
Windows
East
Managed
```

Notice the use of the escape character (a backtick) and a "t" to define the tab character. That had to be in double quotes so that the escape character would be recognized.

The resulting array has four elements, and you can access them individually by using their index numbers:

```
PS C:\> $array[0]
Server1
```

## 24.3 String manipulation

Suppose you have a string of text, and you need to convert it to all uppercase letters. Or perhaps you need to get the last three characters from the string. How would you do it?

In PowerShell, strings are objects, and they come with a great many methods. Remember that a method is simply a way of telling the object to do something, usually to itself, and that you can discover the available methods by piping the object to `Gm`:

```
PS C:\> "Hello" | gm

   TypeName: System.String

Name              MemberType            Definition
----              ----------            ----------
Clone             Method                System.Object Clone()
CompareTo         Method                int CompareTo(System.Object value...
Contains          Method                bool Contains(string value)
CopyTo            Method                System.Void CopyTo(int sourceInde...
EndsWith          Method                bool EndsWith(string value), bool...
Equals            Method                bool Equals(System.Object obj), b...
GetEnumerator     Method                System.CharEnumerator GetEnumerat...
GetHashCode       Method                int GetHashCode()
GetType           Method                type GetType()
GetTypeCode       Method                System.TypeCode GetTypeCode()
IndexOf           Method                int IndexOf(char value), int Inde...
IndexOfAny        Method                int IndexOfAny(char[] anyOf), int...
Insert            Method                string Insert(int startIndex, str...
IsNormalized      Method                bool IsNormalized(), bool IsNorma...
LastIndexOf       Method                int LastIndexOf(char value), int ...
LastIndexOfAny    Method                int LastIndexOfAny(char[] anyOf),...
Normalize         Method                string Normalize(), string Normal...
PadLeft           Method                string PadLeft(int totalWidth), s...
PadRight          Method                string PadRight(int totalWidth), ...
Remove            Method                string Remove(int startIndex, int...
Replace           Method                string Replace(char oldChar, char...
```

```
Split              Method                 string[] Split(Params char[] sepa...
StartsWith         Method                 bool StartsWith(string value), bo...
Substring          Method                 string Substring(int startIndex),...
ToCharArray        Method                 char[] ToCharArray(), char[] ToCh...
ToLower            Method                 string ToLower(), string ToLower(...
ToLowerInvariant   Method                 string ToLowerInvariant()
ToString           Method                 string ToString(), string ToStrin...
ToUpper            Method                 string ToUpper(), string ToUpper(...
ToUpperInvariant   Method                 string ToUpperInvariant()
Trim               Method                 string Trim(Params char[] trimCha...
TrimEnd            Method                 string TrimEnd(Params char[] trim...
TrimStart          Method                 string TrimStart(Params char[] tr...
Chars              ParameterizedProperty  char Chars(int index) {get;}
Length             Property               System.Int32 Length {get;}
```

Some of the more useful `String` methods include the following:

- `IndexOf()` tells you the location of a given character within the string.

  ```
  PS C:\> "SERVER-R2".IndexOf("-")
  6
  ```

- `Split()`, `Join()`, and `Replace()` operate similarly to the `-split`, `-join`, and `-replace` operators I described in the previous section. I tend to use the PowerShell operators rather than the `String` methods.
- `ToLower()` and `ToUpper()` convert the case of a string.

  ```
  PS C:\> $computername = "SERVER17"
  PS C:\> $computername.tolower()
  server17
  ```

- `Trim()` removes white space from both ends of a string; `TrimStart()` and `TrimEnd()` remove white space from the beginning or end of a string.

  ```
  PS C:\> $username = "    Don"
  PS C:\> $username.Trim()
  Don
  ```

All of these `String` methods are great ways to manipulate and modify `String` objects. Note that all of these methods can be used with a variable that contains a string (as in the `ToLower()` and `Trim()` examples), or they can be used directly with a static string (as in the `IndexOf()` example).

## 24.4 Date manipulation

Like `String` objects, `Date` (or `DateTime`, if you prefer) objects come with a great many methods that allow date and time manipulation and calculation:

```
PS C:\> get-date | gm

   TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(System.TimeSpan ...
AddDays              Method         System.DateTime AddDays(double value)
AddHours             Method         System.DateTime AddHours(double value)
```

```
AddMilliseconds      Method         System.DateTime AddMilliseconds(doub...
AddMinutes           Method         System.DateTime AddMinutes(double va...
AddMonths            Method         System.DateTime AddMonths(int months)
AddSeconds           Method         System.DateTime AddSeconds(double va...
AddTicks             Method         System.DateTime AddTicks(long value)
AddYears             Method         System.DateTime AddYears(int value)
CompareTo            Method         int CompareTo(System.Object value), ...
Equals               Method         bool Equals(System.Object value), bo...
GetDateTimeFormats   Method         string[] GetDateTimeFormats(), strin...
GetHashCode          Method         int GetHashCode()
GetType              Method         type GetType()
GetTypeCode          Method         System.TypeCode GetTypeCode()
IsDaylightSavingTime Method         bool IsDaylightSavingTime()
Subtract             Method         System.TimeSpan Subtract(System.Date...
ToBinary             Method         long ToBinary()
ToFileTime           Method         long ToFileTime()
ToFileTimeUtc        Method         long ToFileTimeUtc()
ToLocalTime          Method         System.DateTime ToLocalTime()
ToLongDateString     Method         string ToLongDateString()
ToLongTimeString     Method         string ToLongTimeString()
ToOADate             Method         double ToOADate()
ToShortDateString    Method         string ToShortDateString()
ToShortTimeString    Method         string ToShortTimeString()
ToString             Method         string ToString(), string ToString(s...
ToUniversalTime      Method         System.DateTime ToUniversalTime()
DisplayHint          NoteProperty   Microsoft.PowerShell.Commands.Displa...
Date                 Property       System.DateTime Date {get;}
Day                  Property       System.Int32 Day {get;}
DayOfWeek            Property       System.DayOfWeek DayOfWeek {get;}
DayOfYear            Property       System.Int32 DayOfYear {get;}
Hour                 Property       System.Int32 Hour {get;}
Kind                 Property       System.DateTimeKind Kind {get;}
Millisecond          Property       System.Int32 Millisecond {get;}
Minute               Property       System.Int32 Minute {get;}
Month                Property       System.Int32 Month {get;}
Second               Property       System.Int32 Second {get;}
Ticks                Property       System.Int64 Ticks {get;}
TimeOfDay            Property       System.TimeSpan TimeOfDay {get;}
Year                 Property       System.Int32 Year {get;}
DateTime             ScriptProperty System.Object DateTime {get=if ((& {...
```

Note that the properties enable you to access just a portion of a `DateTime`, such as the day, year, or month:

```
PS C:\> (get-date).month
10
```

The methods enable two things: calculations, and conversions to other formats. For example, to get the date for 90 days ago, I like to use `AddDays()` with a negative number:

```
PS C:\> $today = get-date
PS C:\> $90daysago = $today.adddays(-90)
PS C:\> $90daysago

Saturday, July 24, 2010 11:26:08 AM
```

The methods whose name start with "To" are designed to provide dates and times in an alternative format, such as a short date string:

```
PS C:\> $90daysago.toshortdatestring()
7/24/2010
```

These methods all use your computer's current regional settings to determine the correct way of formatting dates and times.

## 24.5 Dealing with WMI dates

WMI tends to store date and time information in difficult-to-use strings. For example, the `Win32_OperatingSystem` class tracks the last time a computer was started, and the date and time information looks like this:

```
PS C:\> get-wmiobject win32_operatingsystem | select lastbootuptime

lastbootuptime
--------------
20101021210207.793534-420
```

PowerShell's designers knew you wouldn't be able to easily use this information, so they added a pair of conversion methods to every WMI object. Pipe any WMI object to `Gm` and you can see those methods at or near the end:

```
PS C:\> get-wmiobject win32_operatingsystem | gm

   TypeName:
     System.Management.ManagementObject#root\cimv2\Win32_OperatingSystem

Name                                      MemberType     Definition
----                                      ----------     ----------
Reboot                                    Method         System.Management...
SetDateTime                               Method         System.Management...
Shutdown                                  Method         System.Management...
Win32Shutdown                             Method         System.Management...
Win32ShutdownTracker                      Method         System.Management...
BootDevice                                Property       System.String Boo...
...
PSStatus                                  PropertySet    PSStatus {Status,...
ConvertFromDateTime                       ScriptMethod   System.Object Con...
ConvertToDateTime                         ScriptMethod   System.Object Con...
```

I've cut out most of the middle of this output so that you can easily find the `ConvertFromDateTime()` and `ConvertToDateTime()` methods. In this case, what we start with is a WMI date and time, and we want to convert to a normal date and time, so we'd do it like this:

```
PS C:\> $os = get-wmiobject win32_operatingsystem
PS C:\> $os.ConvertToDateTime($os.lastbootuptime)

Thursday, October 21, 2010 9:02:07 PM
```

If you want to make that date and time information part of a normal table, you can use `Select-Object` or `Format-Table` to create custom, calculated columns and properties:

```
PS C:\> get-wmiobject win32_operatingsystem | select BuildNumber,__SERVER,@{
l='LastBootTime';e={$_.ConvertToDateTime($_.LastBootupTime)}}

BuildNumber                __SERVER                 LastBootTime
-----------                --------                 ------------
7600                       SERVER-R2                10/21/2010 9:02:07 PM
```

# 25 Final exam: tackling an administrative task from scratch

Congratulations! You've completed all of the main chapters in this book, and you're ready to put your new knowledge to use. I find that having a practical task in front of you is a great way to cement newly acquired skills, and this chapter's sole purpose is to give you that task.

## 25.1 Tips before you begin

Before you get started, however, I want to offer a few tips.

- You probably *will* get stuck. I almost always do. Don't be afraid to ask for help! You can use the community forums at www.PoshComm.org, or you can log into my own forums at http://connect.ConcentratedTech.com (if you're asked where you took a class with me, just answer "Lunches book" and you'll be allowed in). Manning (www.manning-sandbox.com/forum.jspa?forumID=723) also has a forum dedicated to this book where you can post questions, and I try to monitor those a couple of times a week.
- Spend some time breaking the task down to its main components. Figure out which parts of the task involve the real functionality, and focus on writing commands that create the desired output.
- Once you have completed the necessary commands, you can worry about writing functions and other structures around those commands.

## 25.2 Lab

This lab is going to be a bit different. I'm going to specify a number of criteria, and your job is to create the final result. I'll walk you through the solution in the next section, rather than sending you to MoreLunches.com for the sample solution.

Your job is to create a script module that contains a function named `Get-OSInfo`. You want this function to produce output that includes a computer name, operating-system build number, BIOS serial number, and the last boot date and time for the operating system. The final result should look something like this:

```
ComputerName        OSBuild        BIOSSerial          LastBoot
------------        -------        ----------          --------
localhost           7600           VMware-56 4d 45 ... 10/21/2010 9:02:...
server-r2           7600           VMware-56 4d 45 ... 10/21/2010 9:02:...
```

> **TRY IT NOW** Start by writing a simple script that just produces this output. Don't worry about parameters; hardcode the server names for now. Don't worry about functions and other structure now, either—just get this output to appear on the screen.

Your function should be written as an advanced function (a script cmdlet, if you prefer). It should have two parameters, `-computerName` and `-logFile`. You should be able to execute the function by using any of these patterns (assuming, of course, that SERVER-R2 is a valid computer name):

```
PS C:\> 'localhost','server-r2' | Get-OSInfo
PS C:\> Get-OSInfo -computername 'localhost'
PS C:\> Get-OSInfo -host 'localhost','server-r2'
```

If the `-logFile` parameter isn't supplied, no log should be created. But if the parameter is supplied with a path and filename, your function should delete any existing file of that name when it first runs, and then write any computer names that could not be contacted to that file.

Here are some additional criteria:

- Include comment-based help that describes how `Get-OSInfo` works, including examples.
- When someone imports your module, they should only see `Get-OSInfo` and an alias, `goi`. No other contents of the module should be visible to the user.
- You'll need to query Win32_OperatingSystem and Win32_BIOS. But if the first of those queries fails, you should not even try to perform the second query.
- The last boot time should be displayed as a human-readable date and time.
- If the function is run without providing a `-computerName` parameter, the shell should automatically prompt for a parameter value. The function shouldn't run if `-computerName` isn't supplied.

> **TRY IT NOW** Stop reading here, and see what you can accomplish. Chapters 20 and 22, in particular, should provide some helpful pointers, if you need to refer back to them.

Remember that you don't need to construct this all at once. Instead, start small, with a command or two that accomplishes the core tasks of retrieving the information you need. Then start to build a structure around those commands that will provide the other capabilities, such as parameters, prompting, help, and so forth.

## 25.3 Lab solution

There are many ways to accomplish this task in PowerShell, but listing 25.1 shows how I chose to solve this task. Note that I saved this as /Documents/WindowsPowerShell/Modules/MyModule/MyModule.psm1, and I imported it into the shell by running `Import-Module MyModule`.

**Listing 25.1 `Get-OSInfo` and its supporting code, in a script module**

```
function GetOSInfo                    ❶
{
    param
    (
        [string]$name,                    ❷
        [string]$logfile
    )
    try {
        $continue = $True                          ❸
        $os = Get-WmiObject Win32_OperatingSystem `
               -computerName $name -ea 'Stop'          ❹
    } catch {
        if ($logFile -ne '') {
            $name | Out-File $logfile -append            ❺
        }
        $continue = $False
    }
    if ($continue) {                     ❻
        $bios = Get-WmiObject Win32_BIOS `             ❼
                 -computername $name
        $obj = New-Object PSObject              ❽
        $obj | Add-Member NoteProperty ComputerName $name
        $obj | Add-Member NoteProperty OSBuild ($os.buildnumber)
        $obj | Add-Member NoteProperty BIOSSerial ($bios.serialnumber)
        $obj | Add-Member NoteProperty LastBoot `
                ($os.ConvertToDateTime($os.lastbootuptime))
        Write-Output $obj                                    ❾
    }
}

<#
.SYNOPSIS
Retrieves key information from the specified computer(s)             ❿
.DESCRIPTION
Get-OSInfo uses WMI to retrieve information from the
Win32_OperatingSystem and Win32_BIOS classes. The result
is a combined object, included translated date/time
information for the computer's most recent restart.
.PARAMETER computername
The computer name, or names, to query.
```

```
.PARAMETER logFailures
Include this parameter to have failed computer names
logged to a file. Specify the filename as the value
for this parameter.
.EXAMPLE
Assuming names.txt contains one computer name per line:
  Get-Content names.txt | Get-OSInfo -log c:\errors.txt
.EXAMPLE
Assuming the ActiveDirectory module is available, this
example retrieves information from all computers in the
domain:
  Get-ADComputer -filter * | Select -expand name |            ⓫
  Get-OSInfo
.EXAMPLE
Just use a single, manually-specified computer:
  Get-OSInfo -computername SERVER-R2
#>
function Get-OSInfo
{
    [CmdletBinding()]                               ⓬
    param
    (
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]        ⓭
        [Alias('host')]                                          ⓮
        [string[]]$computerName,
        [string]$logFile = ''                       ⓯
    )

    BEGIN
    {
        if ($logFile -ne '') {                                     ⓰
            Del -Path $logFile -ErrorAction 'SilentlyContinue'
        }

    }

    PROCESS
    {
        foreach ($name in $computername) {                ⓱
            GetOSInfo $name $logFile                  ⓲
        }
    }
}

New-Alias goi Get-OSInfo                         ⓳

Export-ModuleMember -function Get-OSInfo
Export-ModuleMember -alias goi                         ⓴
```

Here's what's happening, starting at the top of the script:

1 The `GetOSInfo` function is doing the real work ❶. In it, I pass in a single computer name and the log file path ❷.

2 The function sets a `$Continue` variable ❸ that assumes the first WMI query will work. That query uses an `-ErrorAction` of `Stop` ❹, so that if an error occurs the `Catch` block will execute.

3 The `Catch` block logs the computer name to a file if a log file was specified ❺. It also sets the `$Continue` variable to `$False`, so that the second WMI query won't execute ❻.

4 The second WMI query ❼ executes only if the first one succeeded.

5 With both WMI queries complete, I build a custom output object and attach properties to it ❽. Notice that the `LastBootUpTime` property is being converted to a normal `DateTime` by using the built-in `ConvertToDateTime()` method that's attached to all WMI objects.

6 Once completed, the custom object is written to the pipeline ❾.

7 The `Get-OSInfo` function is the one I want users to actually run, so I provide comment-based help ❿. That help includes detailed examples of how to use the function ⓫.

8 The function uses cmdlet-style parameters ⓬, including several parameter attributes for the `-computerName` parameter ⓭, and the alias that will allow the user to use `-host` instead of `-computerName` ⓮.

9 The `$logFile` parameter defaults to an empty string ⓯. I used that in the `GetOSInfo` function to determine whether or not a log path was actually provided.

10 In the `BEGIN` block, I check to see if a log file path was provided ⓰. If one was, I delete any existing file having that name. Because there might not be a file, I specify the `-EA SilentlyContinue` parameter to suppress any errors from this command.

11 It's possible to provide computer names via the `-computerName` (or `-host`) parameter, or via the pipeline. The names will end up in the right variable anyway ⓱, so all I need to do is manually enumerate those values and call the `GetOSInfo` worker function once for each computer name ⓲.

12 I define an alias, `goi`, for the `Get-OSInfo` function ⓳.

13 To hide the `GetOSInfo` worker function, I manually export both the `Get-OSInfo` function and the `goi` alias ⓴. That ensures that only those two items will be visible to someone who imports this module.

To be sure, this is a complex script, but much of the complexity is actually in the structure, not in the commands. The underlying commands that are doing much of the work are fairly straightforward. The structure serves to make this more accessible to less-experienced co-workers and colleagues.

# 26 Beyond the operating system: taking PowerShell further

I know you've seen a lot of `Get-Process` and `Get-Service` in this book. There's a reason for that: as I explained toward the beginning of the book, I'm guaranteed of you having access to those cmdlets because they're built into the base shell. Although we also used `Get-WmiObject`, `Get-Hotfix`, and a few other core cmdlets, I like `Get-Service` and `Get-Process` because they exhibit almost all of the possible characteristics of a cmdlet. You can use them to master parameters, pipeline parameter binding, and many other key concepts. Once you've done so, using any other cmdlet is easy: just read its help file to learn about its parameters, and you're good to go.

In this chapter, I want to take a brief look at how you can apply those core skills to the cmdlets that come with other products. This isn't intended to be a tutorial on those products' cmdlets. Rather, I want this to prove to you that, rather than me giving you fish to eat, I've taught you how to fish. In other words, you're prepared to go out and learn additional cmdlets on your own, without much more help from me.

## 26.1 Everything you've learned works the same everywhere

The neat thing about PowerShell is that it forces cmdlet developers into a set of fairly strict patterns. That's not to say every cmdlet developer does a good job of sticking with those patterns, but the nature of PowerShell makes it difficult for them to stray too far from what Microsoft intended for PowerShell to be. That means every cmdlet tends to work more or less like every other cmdlet. All you need to do is read the help, know how to interpret it, and know a few key skills like parenthetical commands and pipeline parameter binding—all of which you've learned in this book.

This chapter is going to be a bit different for me to write, and that's going to make it a bit different for you to read. I've deliberately held off learning any of the SharePoint Server 2010 cmdlets, and I've never even seen the VMware cmdlets. So I'm going to write this chapter as a kind of stream-of-consciousness narrative, meaning that I'll be exploring these cmdlets for the first time, and you're coming along for the ride. Let's see if those core PowerShell skills you've learned in the previous two dozen chapters are sufficient to figure out how these cmdlets work.

## 26.2 SharePoint Server 2010

The first thing you need to do is to set yourself a task of some kind. I've decided that I want to get a list of every Web in every SharePoint site. I'm not going to be using the SharePoint-specific PowerShell shortcut; I regard that as cheating because it will preload the SharePoint commands for me. I'm going to start in the basic shell.

First, I'll see if I can find a SharePoint module or snap-in:

```
PS C:\> get-module -listavailable

ModuleType   Name                       ExportedCommands
----------   ----                       ----------------
Manifest     ActiveDirectory            {}
Manifest     ADRMS                      {}
Manifest     AppLocker                  {}
Manifest     BestPractices              {}
Manifest     BitsTransfer               {}
Manifest     GroupPolicy                {}
Manifest     PSDiagnostics              {}
Manifest     ServerManager              {}
Manifest     TroubleshootingPack        {}
Manifest     WebAdministration          {}


PS C:\> get-pssnapin -registered


Name        : Microsoft.SharePoint.PowerShell
PSVersion   : 1.0
Description : Register all administration Cmdlets for Microsoft SharePoint
               Server

Name        : SqlServerCmdletSnapin100
PSVersion   : 2.0
Description : This is a PowerShell snap-in that includes various SQL Serve
              r cmdlets.

Name        : SqlServerProviderSnapin100
PSVersion   : 2.0
Description : SQL Server Provider
```

There are no modules, but there's a snap-in. I can load that:

```
PS C:\> add-pssnapin microsoft.sharepoint.powershell
```

And I can get a list of commands:

```
gcm -pssnapin microsoft.sharepoint.powershell
```

And the list is huge. I can see that every cmdlet noun starts with “SP,” so let’s see if I can find something to do with sites:

```
PS C:\> gcm -name *site*

CommandType     Name                              Definition
-----------     ----                              ----------
Cmdlet          Add-SPSiteSubscriptionFeat...     Add-SPSiteSubscriptionFea...
Cmdlet          Add-SPSiteSubscriptionProf...     Add-SPSiteSubscriptionPro...
Cmdlet          Backup-SPSite                     Backup-SPSite [-Identity]...
Cmdlet          Clear-SPSiteSubscriptionBu...     Clear-SPSiteSubscriptionB...
Application     dssite.msc                        C:\Windows\system32\dssit...
Cmdlet          Export-SPSiteSubscriptionB...     Export-SPSiteSubscription...
Cmdlet          Export-SPSiteSubscriptionS...     Export-SPSiteSubscription...
Cmdlet          Get-SPEnterpriseSearchQuer...     Get-SPEnterpriseSearchQue...
Cmdlet          Get-SPEnterpriseSearchQuer...     Get-SPEnterpriseSearchQue...
Cmdlet          Get-SPEnterpriseSearchQuer...     Get-SPEnterpriseSearchQue...
Cmdlet          Get-SPEnterpriseSearchSite...     Get-SPEnterpriseSearchSit...
Cmdlet          Get-SPSite                        Get-SPSite [-Limit <Strin...
...
```

This is still a huge list of commands (I’ve truncated it here) because I wasn’t specific about them being cmdlets. But I found `Get-SPSite`. Time to read the help.

```
PS C:\> help get-spsite -full
```

I always start with the full help, not the abbreviation, because I also want to review the parameters and see the examples. I’m seeing some text that concerns me:

> … every site collection returned by the `Get-SPSite` cmdlet is automatically disposed of at the end of the pipeline. To store the results of `Get-SPSite` in a local variable, the `Start-SPAssignment` and `Stop-SPAssignment` cmdlets must be used to avoid memory leaks.

Sounds awful, so let’s not do the variable thing unless we have to. It looks like I can just run `Get-SPSite` and get a list of sites, so let’s try that:

```
PS C:\> get-spsite

Url
---
http://server-r2
http://server-r2/my
```

Perfect. Now, I want to get the Webs from those sites. I saw in the `Get-SPSite` help examples that there’s a cmdlet called `Get-SPWeb`, so I’ll read its full help. It says I need to use an `-Identity` parameter, which can be a full URL or a relative path. It doesn’t accept pipeline input, though. Its `-Site` parameter accepts pipeline input; unfortunately the help doesn’t say if it accepts ByValue or ByPropertyName. How annoying.

But it does say that the parameter "specifies the URL or GUID of the site collection from which to list subsites."

Well, I know where to get a URL—it was in the output of the previous command I ran. The `-AssignmentCollection` parameter also accepts pipeline input, which isn't important to me right now, but I'll make a note of it. Its description talks about things using large amounts of memory and memory management, so I'll come back and read it later.

Right now, I'll run this:

```
PS C:\> get-spsite | get-spweb | gm


   TypeName: Microsoft.SharePoint.SPWeb

Name                                  MemberType Definition
----                                  ---------- ----------
AddApplicationPrincipal               Method     Microsoft.SharePoint.S...
AddProperty                           Method     System.Void AddPropert...
AddSupportedUICulture                 Method     System.Void AddSupport...
...
```

There is more to see, but it's too long to paste in here. But I now know that I can pipe `Get-SPSite` to `Get-SPWeb` to get the Webs for those sites, and `Gm` is telling me what properties are available for me to work with once I've got the Webs. I'll pick out a few interesting-looking properties and try to view them:

```
PS C:\> get-spsite | get-spweb | ft Url,Title,IsRootWeb

Url                     Title                                     IsRootWeb
---                     -----                                     ---------
http://server-r2        Nugget Lab                                     True
http://server-r2/my     My Site                                        True
```

Hopefully by now you're comfortable with `Ft`, the alias for `Format-Table`, so this command should make sense. There are a ton of methods on the Web objects, too, all of which can make one of those Webs do something for me. I don't know if there are equivalent cmdlets or not, but I can use `ForEach-Object` to execute methods.

For example, there's a `Delete()` method, so I bet I could run the following command to get my boss to fire me:

```
PS C:\>get-spsite | get-spweb | foreach-object { $_.Delete() }
```

Keep in mind that methods don't support `-confirm` or `-whatif`, and you always have to have the parentheses after the method name, even if they don't contain any parameters.

Browsing the list of other SharePoint-related commands, I can see a lot of capabilities. I should be able to do just about anything. The trick to learning how to accomplish any task involves these three steps:

- Discovering available commands
- Reading commands' help to figure out how they work, paying close attention to pipeline parameter binding options and to usage examples

- Experimenting in a test environment (like a virtual machine) to avoid breaking anything

Mix in a little Google or Bing searching on the side, and you should be able to figure out any of it.

## 26.3 VMware vSphere and vCenter

VMware vSphere, including its vCenter management tool, is a product I've never even used before. I work with several ESXi servers, but I haven't ever been called upon to do any automation with VMware. Their PowerShell extensions are based on the same models as their other scripting toolkits for VBScript and so on, and I've heard that they're not quite as well-structured as the cmdlets Microsoft has produced for their products. We'll see! Again, the goal here isn't to provide you with a tutorial on these cmdlets (I'm not sure I'm qualified to do so), but to help you see how I tackle a completely unknown set of cmdlets and teach myself to do at least a basic task.

As usual, I start by installing the cmdlets (which are in VMware's VI Toolkit), and adding the snap-in into PowerShell. I'm looking to do a simple inventory of VM configuration settings, so I run this command:

```
PS C:\>Get-Command -verb get -noun *config*
```

I use `config` instead of `configuration` in case the folks who developed these cmdlets were using abbreviations and shortened word forms.

One of the cmdlets that pops out at me is `Get-VMResourceConfiguration`—excellent! I run it with a single virtual machine on my host and see this:

```
PS C:\> Get-VMResourceConfiguration MyVM

VirtualMachineId           : VirtualMachine-vm-3674
NumCpuShares               : 1000
CpuReservationMhz          :0
CpuLimitMhz                : -1
CpuSharesLevel             : Normal
NumMemShares               : 5120
MemReservationMB           : 0
MemLimitMB                 : -1
MemSharesLevel             : Normal
DiskResourceConfiguration  : {2000}
HTCoreSharing              : Any
CpuAffinity                : NoAffinity
```

That's exactly what I was looking for. Browsing through the list of cmdlets that have a noun prepended with "VM," I also see `Invoke-VMScript`, which—according to the help—appears to be a way to inject a script or command directly into a virtual machine. It requires that PowerShell be installed within each VM, which is fine by me.

This is an excellent extension to PowerShell's own remoting, because it lets me target virtual machines without needing to know their computer names. The command runs through the VMware Tools link, so I don't even need to enable regular PowerShell remoting.

These are just two quick examples of how easily I was able to discover useful commands, simply by loading the snap-in, exploring the commands that were added, and reading their help. If you're interested in a more complete guide to the VMware cmdlets, check out *Managing VMware Infrastructure with Windows PowerShell: TFM* by Hal Rottenberg (http://www.sapienpress.com/vmware.asp).

## 26.4 Third-party Active Directory management

I want to wrap up this chapter with a quick note about managing Active Directory. In this book, I've used Microsoft's AD cmdlets, which ship with Windows Server 2008 R2 and are available in the Remote Server Administration Tools (RSAT) for Windows 7 and later. A lot of experienced PowerShell/AD gurus don't like the Microsoft cmdlets for a number of reasons, one of which is their inability to access schema extensions, Terminal Services attributes, and so forth. I use them because they're a great example of how to use pipeline parameter binding (that's why I used them in chapter 7, for example). But I readily acknowledge their shortcomings. If you're a serious AD administrator, you'll want to check out Quest Software (http://quest.com/powershell) and download their free PowerShell Commands for Active Directory. These don't require any additional software on your domain controllers (the Microsoft ones do on DCs prior to Win2008R2), and they'll even work with Win2000-based domains if you still have any of those lying around. They also work with Active Directory Lightweight Directory Services (AD LDS), where the Microsoft cmdlets won't.

These Quest cmdlets are well written, although they initially lacked the same rich pipeline parameter binding of the Microsoft cmdlets. Originally you couldn't do the same `Import-CSV` trick that I did in chapter 7, although you could do something similar.

For example, assuming you started with a CSV file like this one,

```
LoginName,Department,City,Title,FirstName,LastName
DonJ,IT,Las Vegas,CIO,Don,Jones
GregS,Janitorial,Denver,Custodian,Greg,Shields
JeffH,IT,Syracuse,Technician,Jeffery,Hicks
ChrisG,Finance,Las Vegas,Accountant,Christopher,Gannon
```

you could run a command like this:

```
Import-CSV c:\users.csv | ForEach-Object {
➥ New-QADUser -name $_.LoginName -sAMAccountName $_.LoginName
➥ -department $_.department -city $_.city -title $_.title
➥ -sn $_.lastname -givenname $_.firstname }
```

It's a lot more typing, but it gets the job done. Notice that Quest uses an additional "Q" prefix on the nouns of their cmdlets, nicely distinguishing between Microsoft's `Get-ADUser` and Quest's `Get-QADUser`.

As of v1.0.6 of Quest's cmdlets, however, parameter binding has been added, so provided your CSV column names match the parameter names, you could just do this:

```
Import-CSV users.csv | New-QADUser -import
```

You have to add that `-import` parameter to make this work, which is a little inconsistent with the way most cmdlets work, but at least you can do it.

Of course, in the preceding example CSV file, the column names don't match up—so you can still use the technique that I did to attach the right CSV columns to the parameters. You could also use property renaming, the technique I showed you in chapter 7.

This is probably a nice time to point out that, in PowerShell, there are usually twenty ways to accomplish anything. Right here, I've suggested three ways in which you could use `New-QADUser` to import new users from a CSV file. Exactly which technique you choose is constrained a bit by your CSV file having the right column names or not. You might also, in your searches of the internet, see something like this:

```
$OuBorn = 'OU=PowerShell,DC=cp2,DC=mosel'
$Freshmen = 'E:\powershell\QAD\bunch4.csv'
import-csv $Freshmen |`
where {new-QADUser -ParentContainer $OuBorn `
-name $_.name -sAMAccountName $_.sAMAccountName}
```

Let's break that down a bit:

- The first two lines define variables, which will be the OU that the new users are created in, and the location of the CSV file. There's no reason to put those static values into a variable when you can specify them as part of the commands, but using variables doesn't hurt.
- The next line imports the CSV file. This line ends in a backtick, which is commonly seen but unnecessary. The author's intent here is to escape the carriage return, which prevents the shell from executing the line and lets you break a long line into multiple physical lines. It's unnecessary because the line already ends in a pipe character, which tells the shell that you've got more to type.
- Rather than piping the CSV objects to `ForEach-Object`, this author is piping them to `Where-Object`. I find that weird, but you see it a *lot* out there in the real world. The net effect is the same, because `Where-Object` will execute that script block once for each object piped in, but it's not what `Where-Object` is really intended for. The backtick at the end of the fourth line is necessary if you want to break the line at that point.

In my experience, Quest's cmdlets are definitely used in more production environments than Microsoft's cmdlets. In the end, you should probably look at both, and use whichever one makes sense for the task at hand. There's nothing stopping you from having both installed on a computer, or even from having both loaded into the shell at the same time.

# 27 Never the end

We've come to the end of this book, but it's hardly the end of your PowerShell exploration. There's a lot more in the shell to learn (hmm, perhaps an "Advanced" book is in order?), and based on what you've learned here, you'll be able to teach yourself much of it (so much for the book idea, I guess). This short chapter will help point you in the right directions.

## 27.1 Ideas for further exploration

There's a lot more that you can do in PowerShell. We've really only scratched the surface, although you should certainly have learned enough in this book to be very, very effective. Here are some of the other things you might want to explore:

- Create your own predefined views. There's a pretty simple XML format for doing so, and the `Update-FormatData` command loads views into the shell once you're done.
- Work with XML-formatted data, using PowerShell's `[xml]` type.
- Access data in a database. This requires you to use a few raw .NET Framework classes, but there are simple, copy-and-paste patterns you can rely on to get the job done.
- Write internationalized scripts that can substitute strings in different languages. This is especially helpful if you have colleagues in other countries who don't speak English as a first language.
- Access Component Object Model (COM) objects. This provides access to a wide range of functionality that's been in Windows since pretty much the beginning.

- Use transactional operations. As I'm writing this, only the registry supports transactional operations, but they enable you to conduct many operations and then have them all commit as a single unit.
- Create a graphical user interface (GUI) from within your script. Tools like SAPIEN PrimalForms can assist with this, but you can also manually access the .NET Framework's Windows Forms and Windows Presentation Foundation (WPF) technologies to create dialog boxes and other GUI elements.
- PowerShell includes rich support for regular expressions, which enable you to describe a text pattern to a computer (like a UNC path or IP address) and have the computer match that pattern inside text data.

Obviously, all of these are beyond the scope of this book, but you'll find many of them covered by the additional resources that I list at the end of this chapter. This list should also provide you with the keywords to punch into a search engine to get you started.

## 27.2 "Now that I'm done, where do I start?"

The best thing to do now is to pick a task. Choose something in your production world that you personally find repetitive, and automate it using the shell. You'll almost certainly run across things that you don't know how to do, and that's the perfect place to start learning.

Here are some of the things I've seen other administrators tackle:

- Write a script that changes the password a service uses to log in, and have it target multiple computers that are running that service. (Actually, you could do this in a single command.)
- Write a script that automates new user provisioning, including creating user accounts, mailboxes, and home directories. Setting NTFS permissions with PowerShell is tricky, but consider using a tool like Cacls.exe or Xcacls.exe from within your PowerShell script, instead of PowerShell's native (and complex) `Get-ACL` and `Set-ACL` cmdlets.
- Write a script that manages Exchange mailboxes in some way—perhaps getting reports on the largest mailboxes, or creating charge-back reports based on mailbox sizes.
- Automate the provisioning of new websites in IIS, using the WebAdministration module included in Windows Server 2008 R2 (which also works against IIS 7 in Windows Server 2008).

The biggest thing to remember is to *not over-think it*. I once met an administrator who struggled for weeks to write a robust file-copying script in PowerShell so that he could deploy content across a web server farm. "Why not just use Xcopy or Robocopy?" I asked. He stared at me for a minute, and then laughed. He'd gotten so wrapped up in "doing it in PowerShell" that he forgot that PowerShell can use all of the excellent utilities that are already out there.

## 27.3 Other resources you'll grow to love

I spend a lot of time working with, writing about, and teaching PowerShell. Ask my family—sometimes I barely shut up about it long enough to eat dinner. That means I've accumulated a lot of online resources that I use daily, and that I recommend to all of my students. Hopefully they'll provide you with a good starting point, as well.

- MoreLunches.com—This should be your first stop, if you haven't already bookmarked the site. There you'll find free bonus and companion content for this book, including the lab answers, video demonstrations, bonus articles, and additional recommended resources. You'll also be able to download the longer code listings for this book, so that you don't have to type them in manually. Consider bookmarking the site and visiting often to refresh what you've learned in this book.
- http://WindowsITPro.com/go/DonJonesPowerShell—This is a landing page for my online Frequently Asked Questions (FAQ) and blog about Windows PowerShell. You'll also find bimonthly feature articles. The layout of the page changes from time to time, so if you have trouble finding the blog articles, go directly to the blog index at http://www.windowsitpro.com/blogs/PowerShellwithaPurpose.aspx. I post a new blog article at least twice weekly, and they're always either tutorials, tips, or PowerShell-related product reviews.
- http://Connect.ConcentratedTech.com—This is a private discussion forum for past students—and that now includes you. You'll need to register for an account, but once you do, you're welcome to post your PowerShell questions and I'll do my best to answer. I also monitor a forum hosted by Manning, http://www.manning-sandbox.com/forumindex.jspa, if you'd prefer to use that.
- http://ShellHub.com—This is a website that I maintain. It's a handpicked list of other PowerShell-related online resources, including the blogs I read most, third-party PowerShell tools, and more. Pretty much every URL I've ever recommended to someone is listed here. In the event that any other URL I give you changes, you can hop on ShellHub.com to find an update.

Students often ask if there are any other PowerShell books that I recommend. There are only a few that I keep right on my desk.

- One is *Windows PowerShell v2.0: TFM*, published by SAPIEN Press. I coauthored this with Jeffery Hicks, so I'm a bit biased, but it covers almost every single thing an administrator can do with PowerShell, including numerous full-length examples. The examples can also be downloaded from http://SAPIENPress.com (navigate to the book's page and scroll all the way to the bottom). Some of the content in the book gets pretty advanced, but it's a great reference.

- I also refer to *Windows PowerShell in Action* (also published by Manning) a lot. Written by Bruce Payette, the lead developer for PowerShell, the book isn't so much a tutorial (like this one), but more of a brain-dump on how and why the shell works. You'll learn a lot, and it's a good way to understand many of the gotchas that you run across as you're learning the shell.
- Finally, Richard Siddaway's *PowerShell in Practice* is a "cookbook" approach to PowerShell, offering a number of ready-made solutions to common tasks.

Finally, if you'd like some full-length video-based training for PowerShell, visit http://shellhub.com/training.php for suggestions. Keep in mind, though, that MoreLunches.com hosts free companion video content for each chapter in this book!

# 28 PowerShell cheat sheet

This is my opportunity to assemble a lot of the little gotchas into a single place. If you're ever having trouble remembering what something is or does, flip to this chapter first.

## 28.1 Punctuation

There's no doubt that PowerShell is full of punctuation, and much of it has a different meaning in the help files than it does in the shell itself. Here's what it all means within the shell:

- `` ` `` *(backtick)*—This is PowerShell's escape character. It removes the special meaning of any character that follows it. For example, a space is normally a separator, which is why `cd c:\Program Files` generates an error. Escaping the space, ``cd c:\Program` Files``, removes that special meaning and forces the space to be treated as a literal, so the command works.
- `~` *(tilde)*—When used as part of a path, this represents the current user's home directory, as defined in the `UserProfile` environment variable.
- `()` *(parentheses)*—These are used in a couple of ways:
  - Just as in math, parentheses define an order of execution. PowerShell will execute parenthetical commands first, from the innermost parentheses to the outermost. This is a good way to run a command and have its output feed the parameter of another command: `Get-Service -computerName (Get-Content c:\computernames.txt)`
  - Parentheses also enclose the parameters of a method, and they must be included even if the method doesn't require any parameters: `ChangeStartMode('Automatic')`, for example, or `Delete()`.

- `[ ]` *(square brackets)*—These have two main uses in the shell:
  - They contain the index number when you want to refer to a single object within an array or collection: `$services[2]` gets the third object from `$services` (indexes are always zero-based).
  - They contain a data type when you're casting a piece of data as a specific type. For example, `$myresult / 3 -as [int]` casts the result as a whole number (integer), and `[xml]$data = Get-Content data.xml` will read the contents of Data.xml and attempt to parse it as a valid XML document.
- `{ }` *(curly braces or curly brackets)*—These have three uses:
  - They contain blocks of executable code or commands, called *script blocks*. These are often fed to parameters that expect a script block or a filter block: `Get-Service | Where-Object { $_.Status -eq 'Running' }`
  - They contain the key-value pairs that make up a new hashtable. The opening brace is always preceded by an `@` sign. Notice that in this example I'm using braces both to enclose the hashtable key-value pairs (of which there are two) and to enclose an expression script block, which is the value for the second key, "e":
    `$hashtable = @{l='Label';e={expression}}`
  - When a variable name contains spaces, braces must surround the name:
    `${My Variable}`
- `' '` *(single quotation marks)*—These contain string values. PowerShell doesn't look for the escape character, nor does it look for variables, inside single quotes.
- `" "` *(double quotation marks)*—These contain string values. PowerShell looks for escape characters and the `$` character inside double quotes. Escape characters are processed, and the characters following a `$` symbol (up to the next white space) are taken as a variable name and the contents of that variable are substituted. For example, if the variable `$one` contains the value `World`, then ``$two = "Hello $one `n"`` will contain `Hello World` and a carriage return (`` `n `` is a carriage return).
- `$` *(dollar sign)*—This character tells the shell that the following characters, up to the next white space, represent a variable name. This can be tricky when working with cmdlets that manage variables. Supposing that `$one` contains the value `two`, then `New-Variable -name $one -value 'Hello'` will create a new variable named `two`, with the value `Hello`, because the dollar sign tells the shell that you want to use the contents of `$one`. `New-Variable -name one -value 'Hello'` would create a new variable `$one`.
- `%` *(percent sign)*—This is an alias for the `ForEach-Object` cmdlet.
- `?` *(question mark)*—This is an alias for the `Where-Object` cmdlet.
- `>` *(right angle bracket)*—This is a sort of alias for the `Out-File` cmdlet. It's not technically a true alias, but it does provide for Cmd.exe-style file redirection: `dir > files.txt`.

- `+ - * /` *(math operators)*—These function as standard arithmetic operators. Note that `+` is also used for string concatenation.
- `-` *(dash or hyphen)*—This precedes both parameter names and operators, such as `-computerName` or `-eq`. It also separates the verb and noun components of a cmdlet name, as in `Get-Content`, and serves as the subtraction arithmetic operator.
- `@` *(at sign)*—This has four uses in the shell:
  - It precedes a hashtable's opening curly brace (see *curly braces*, above).
  - When used before parentheses, it encloses a comma-separated list of values that form an array: `$array = @(1,2,3,4)`. But both the `@` sign and the parentheses are optional, because the shell will normally treat any comma-separated list as an array anyway.
  - It denotes a *here-string*, which is a block of literal string text. A here-string starts with `@"` and ends with `"@`, and the closing mark must be on the beginning of a new line. Run `help about_quoting_rules` for more information and examples. Here-strings can also be defined using single quotes.
  - It is PowerShell's splat operator. If you construct a hashtable where the keys match parameter names, and those values' keys are the parameters' values, then you can splat the hashtable to a cmdlet. The B# .NET Blog has a "Windows PowerShell 2.0 Feature Focus—Splat, Split and Join" article that provides a good example of splatting (http://mng.bz/xV7h).
- `&` *(ampersand)*—This is PowerShell's invocation operator, instructing the shell to treat something as a command and to run it. For example, `$a = "Dir"` places the string `"Dir"` into the variable `$a`; `& $a` will run the `Dir` command.
- `;` *(semicolon)*—This is used to separate two independent PowerShell commands that are included on a single line: `Dir ; Get-Process` will run `Dir` and then `Get-Process`. The results are sent to a single pipeline, but the results of `Dir` aren't piped to `Get-Process`.
- `#` *(pound sign or hash mark)*—This is used as a comment character. Any characters following `#`, to the next carriage return, are ignored by the shell. The angle brackets, `<` and `>`, are used as part of the tags that define a block comment: Use `<#` to start a block comment, and `#>` to end one. Everything within the block comment will be ignored by the shell.
- `=` *(equal sign)*—This is the assignment operator, used to assign a value to a variable: `$one = 1`. It isn't used for quality comparisons; use `-eq` instead. Note that the equal sign can be used in conjunction with a math operator: `$var +=5` will add `5` to whatever is currently in `$var`.
- `|` *(pipe)*—The pipe is used to convey the output of one cmdlet to the input of another. The second cmdlet (the one receiving the output) uses pipeline parameter binding to determine which parameter or parameters will actually receive the piped-in objects. Chapter 7 has a discussion of this process.

- `\` *or* `/` *(backslash or slash)*—A forward slash is used as a division operator in mathematical expressions; either the forward slash or backslash can be used as a path separator in file paths: `C:\Windows` is the same as `C:/Windows`. The backslash is also used as an escape character in WMI filter criteria and in regular expressions.
- `.` *(period)*—The period has three main uses:
  - It's used to indicate that you want to access a member, such as a property or method, or an object: `$_.Status` will access the `Status` property of whatever object is in the `$_` placeholder.
  - It's used to *dot source* a script, meaning that the script will be run within the current scope, and anything defined by that script will remain defined after the script completes, for example, `c:\myscript.ps1`.
  - Two dots (`..`) form the range operator, which is discussed later in this chapter. You will also see two dots used to refer to the parent folder in the filesystem, such as in the path `..\`.
- `,` *(comma)*—Outside of quotation marks, the comma separates the items in a list or array: `"One",2,"Three",4`. It can be used to pass multiple static values to a parameter that can accept them: `Get-Process -computername Server1, Server2,Server3`.
- `:` *(colon)*—The colon (technically, two colons) is used to access static members of a class; this gets into .NET Framework programming concepts. `[datetime]::now` is an example (although you could achieve that same task by running `Get-Date`).
- `!` *(exclamation point)*—This is an alias for the `-not` Boolean operator.

I think the only piece of punctuation on a U.S. keyboard that PowerShell doesn't actively use for something is the caret (`^`), although those do get used in regular expressions.

## 28.2 Help file

Punctuation within the help file takes on slightly different meanings:

- `[ ]`—Square brackets that surround any text are indicating that the text is optional. That might include an entire command (`[-Name <string>]`), or it might indicate that a parameter is positional and that the name is optional (`[-Name] <string>`). It can also indicate both: that a parameter is optional, and if used, can be used positionally (`[[-Name] <string>]`). It's always legal to use the parameter name, if you're in any doubt.
- `[ ]`—Adjacent square brackets indicate that a parameter can accept multiple values (`<string[]>` instead of `<string>`).
- `< >`—Angle brackets surround data types, indicating what kind of value or object a parameter expects: `<string>`, `<int>`, `<process>`, and so forth.

Always take the time to read the full help (add `-full` to the `help` command), because it provides maximum detail as well as, in most cases, usage examples.

## 28.3 Operators

PowerShell doesn't use the traditional comparison operators found in most programming languages. Instead, it uses these:

- `-eq`—Equality (`-ceq` for case-sensitive string comparisons)
- `-ne`—Inequality (`-cne` for case-sensitive string comparisons)
- `-ge`—Greater than or equal to (`-cge` for case-sensitive string comparisons)
- `-le`—Less than or equal to (`-cle` for case-sensitive string comparisons)
- `-gt`—Greater than (`-cgt` for case-sensitive string comparisons)
- `-lt`—Less than (`-clt` for case-sensitive string comparisons)
- `-contains`—Returns `True` if the specified collection contains the object specified (`$collection -contains $object`); `-notcontains` is the reverse.

There are logical operators used to combine multiple comparisons:

- `-not`—Reverses `True` and `False` (the `!` symbol is an alias for this operator).
- `-and`—Both subexpressions must be `True` for the entire expression to be `True`.
- `-or`—Either subexpression can be `True` for the entire expression to be `True`.

In addition, there are operators that perform specific functions:

- `-join`—Joins the elements of an array into a delimited string
- `-split`—Splits a delimited string into an array
- `-replace`—Replaces occurrences of one string with another
- `-is`—Returns `True` if an item is of the specified type (`$one -is [int]`)
- `-as`—Casts the item as the specified type (`$one -as [int]`)
- `..`—A range operator; `1..10` returns ten objects, 1 through 10
- `-f`—The format operator, replacing placeholders with values: `"{0}, {1}" -f "Hello","World"`

## 28.4 Custom property and column syntax

In chapters 7 and 8, I showed you how to define custom properties using `Select-Object`, or custom columns and list entries using `Format-Table` and `Format-List` respectively. Here's that hashtable syntax.

You do this for each custom property or column:

```
@{label='Column_or_Property_Name';expression={Value_expression}}
```

Both of the keys, `Label` and `Expression`, can be abbreviated as `l` and `e` respectively (be sure to type a lowercase "L" and not the number 1; you could also use `n` for "Name," in place of the lowercase "L").

```
@{l='Column_or_Property_Name';e={Value_expression}}
```

Within the expression, the `$_` placeholder can be used to refer to the current object (such as the current table row, or the object to which you're adding a custom property):

```
@{l='ComputerName';e={$_.Name}}
```

## 28.5 Pipeline parameter input

Pipeline parameter binding was discussed in chapter 7, where you learned that there are two types of parameter binding: ByValue and ByPropertyName. ByValue occurs first, and ByPropertyName only occurs if ByValue didn't work.

For ByValue, the shell looks at the type of the object that was piped in. You can discover that type name by piping the object to `Gm` yourself. The shell then looks to see if any of the cmdlet's parameters accept that type of input and are configured to accept pipeline input ByValue. It's not possible for a cmdlet to have two parameters binding the same data type in this fashion. In other words, you shouldn't see a cmdlet that has two parameters, each of which accepts `<string>` input, both of which accept pipeline input ByValue.

If ByValue doesn't work, the shell switches to ByPropertyName. Here, it simply looks at the properties of the piped-in object and attempts to find parameters with the exact same names that can accept pipeline input ByPropertyName. So if the piped-in object has properties `Name`, `Status`, and `ID`, the shell will look to see if the cmdlet has parameters named `Name`, `Status`, and `ID`. Those parameters must also be tagged as accepting pipeline input ByPropertyName, which you can see when reading the full help (add `-full` to the `help` command).

Figure 28.1 illustrates this process.

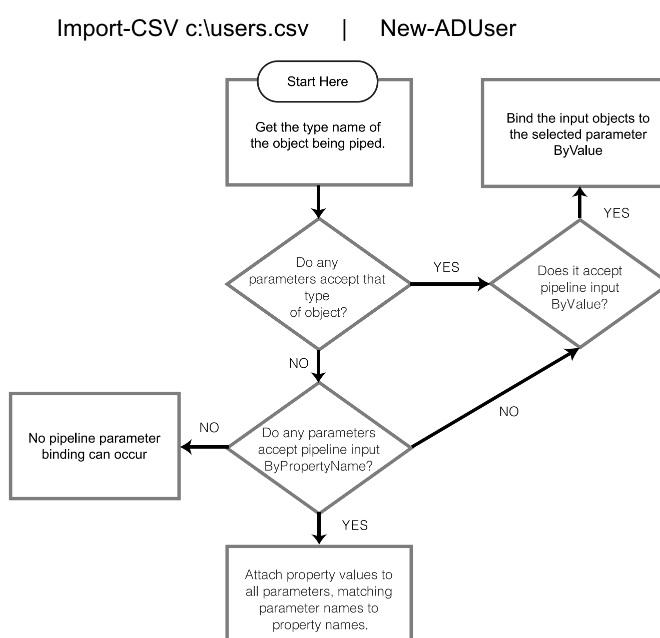



**Figure 28.1 This chart will help you keep track of the pipeline parameter-binding process.**

## 28.6 When to use $_

This is probably one of the most confusing things about the shell: when is the `$_` placeholder permitted?

This placeholder only works when the shell is explicitly looking for it and is prepared to fill it in with something. Generally speaking, that only happens within a script block that's dealing with pipeline input, in which case the `$_` placeholder will contain one pipeline input object at a time. You'll run across this in a few different places:

- In the filtering script block used by `Where-Object`:

  ```
  Get-Service | Where-Object {$_.Status -eq 'Running' }
  ```

- In the script blocks passed to `ForEach-Object`, such as the main `Process` script block typically used with the cmdlet:

  ```
  Get-WmiObject -class Win32_Service -filter "name='mssqlserver'" |
    ForEach-Object -process { $_.ChangeStartMode('Automatic') }
  ```

- In the `Process` script block of a filtering function or an advanced function. Refer to chapter 20 for more information about this.
- In the expression of a hashtable that's being used to create a custom property or table column. Refer to the "Custom property and column syntax" section in this chapter for more details, or read chapters 7 and 8 for a more complete discussion.

In every one of those cases, `$_` occurs only within the curly braces of a script block. That's a good rule to remember for figuring out when it's okay to use `$_`.

# index

## Symbols



## A



## B

## C

## D



## E

## F



## G

## L



## M



## N



## O

## Q



## R

## S

## T



## U



## V

WINDOWS ADMINISTRATION

# *Learn* WINDOWS POWERSHELL IN A MONTH OF LUNCHES

DON JONES

**Free ebook** SEE INSERT

Windows has so many control panels, consoles, APIs, and wizards it's really hard to keep track of all the locations and settings you'll need. PowerShell is a godsend: it provides a single, unified administrative command line. It accepts and executes commands immediately. And it has in-built language features that will let you write scripts to control any Windows component, including servers like Exchange, IIS, and SharePoint.

**Learn Windows PowerShell in a Month of Lunches** is a newly designed tutorial for system administrators. Just set aside one hour a day—lunchtime would be perfect—for a month, and you'll be automating administrative tasks in a hurry. Author Don Jones combines his in-the-trenches experience with a unique teaching style to help you master the effective parts of PowerShell quickly and painlessly.

## What's Inside

- Learn PowerShell 2—no experience required!
- Concise lessons for busy administrators
- Practical examples and techniques in every lesson

## About the Author

**Don Jones** is a PowerShell MVP, speaker, and trainer. He developed the Microsoft PowerShell courseware and has taught PowerShell to more than 20,000 IT pros. Don writes the PowerShell column for TechNet Magazine and blogs for WindowsITPro.com.

"A seminal guide to PowerShell. Highly recommended."
—Ray Booysen, BNP Paribas

"The book I wish I'd had when I started PowerShell."
—Richard Siddaway, Serco

"Tons of useful exercises allow powerful hands-on learning."
—Chuck Durfee
Graebel Companies

"Whether a beginner or an intermediate, you'll need *no other* book."
—David Moravec
PowerShell.cz

For access to the book's forum and a free ebook for owners of this book, go to manning.com/LearnWindowsPowerShellinaMonthofLunches

ISBN-13: 978-1-617290213
ISBN-10: 1-617290211
54499
9 781617 290213

MANNING US $44.99/CAN $51.99 [INCLUDING EBOOK]