Windows PowerShell 3.0

First Steps

Ed Wilson

Windows PowerShell 3.0 First Steps

Get started with this powerful Windows administration tool

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

Discover how to:

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

About the Author

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

ISBN: 978-0-7356-8100-2

U.S.A. $29.99
Canada $31.99
[*Recommended*]

Operating Systems/Windows Server

Microsoft

Microsoft

Windows PowerShell 3.0 First Steps

Ed Wilson

Published with the authorization of Microsoft Corporation by:
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472

ISBN: 978-0-7356-8100-2

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

Printed and bound in the United States of America.

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

Acquisitions and Developmental Editor: Michael Bolinger

Production Editor: Melanie Yarbrough

Editorial Production: Box Twelve Communications

Technical Reviewer: Brian Wilhite

Indexer: Box Twelve Communications

Cover Design: Twist Creative • Seattle

Cover Composition: Ellie Volckhausen

Illustrator: Rebecca Demarest

To Teresa, my soul mate.

—Ed Wilson

Contents at a glance

Contents

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

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

www.microsoft.com/learning/booksurvey/

Foreword

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

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

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

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

No matter how diverse the skill set of my students, there is something for everyone in my classroom. Windows PowerShell has been created in such a way that it can be fun and effective for everyone from the IT novice to the expert developer. For example, the fact that it is

fully object-based and sits on top of the .NET Framework is a detail that pure beginners might have no knowledge of. They can go about their Windows PowerShell days simply running commands, never really digging into the object model, but still implement valuable automation. The day they learn about objects, they can start to unlock so much more. The fact that Windows PowerShell can appeal to such diverse skills levels simultaneously is amazing to me.

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

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

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

Introduction

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

System Requirements

Hardware Requirements

Your computer should meet the following minimum hardware requirements:

- 2.0 GB of RAM (more is recommended)
- 80 GB of available hard disk space
- Internet connectivity

Software Requirements

To complete the exercises in this book, you should have Windows PowerShell 3.0 installed:

- You can obtain Windows PowerShell 3.0 from the Microsoft Download Center by downloading the Windows Management Framework and installing it on either Windows 7 Service Pack 1, Windows Server 2008 R2 SP1, or Windows Server 2008 Service Pack 2.
- Windows PowerShell 3.0 is already installed on Windows 8 and on Windows Server 2012. You can obtain evaulation versions of those operating systems from TechNet:

  http://technet.microsoft.com/en-US/evalcenter/hh699156.aspx?ocid=wc-tn-wctc

  http://technet.microsoft.com/en-US/evalcenter/hh670538.aspx?wt.mc_id=TEC_108_1_4
- The section on Active Directory requires access to Active Directory Domain Services. For those examples, ensure you have access to Windows Server 2012.
- For the chapter on Exchange server, you need access to a server running Microsoft Exchange Server 2013. You can obtain an evaluation version of that from TechNet:

  http://technet.microsoft.com/en-us/evalcenter/hh973395.aspx

Acknowledgments

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

Support & Feedback

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

Errata

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

http://aka.ms/WinPS3FS/errata

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

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

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

We Want to Hear from You

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

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

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

Stay in Touch

Let us keep the conversation going! We are on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 1

Overview of Windows PowerShell 3.0

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

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

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

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

Understanding Windows PowerShell

Windows PowerShell comes in two flavors. The first is an interactive console (similar to a KORN or BASH console in the UNIX world) built into the Windows command prompt. The Windows PowerShell console makes it simple to type short commands and to receive sorted, filtered, and formatted results. These results easily display to the console but also can redirect to .xml, .csv, or text files. The Windows PowerShell console offers several advantages such as speed, low memory overhead, and a comprehensive transcription service that records all commands and command output.

The other flavor of Windows PowerShell is the Windows PowerShell ISE. The Windows PowerShell ISE is an Integrated Scripting Environment, but this does not mean you must use it to write scripts. In fact, many Windows PowerShell users like to write their code in the Windows PowerShell ISE to take advantage of syntax coloring, drop-down lists, and automatic parameter revelation features.

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

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

Working with Windows PowerShell

Windows PowerShell 3.0 is included on Windows 8 and Windows Server 2012. On Windows 8, you need only type the first few letters of the word *PowerShell* in the Start window before Windows PowerShell appears as an option. Figure 1-1 illustrates this point. I typed only **pow** in the Search box before the Start window offered Windows PowerShell as an option.

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

Because navigating to the Start window and typing **pow** each time I want to launch Windows PowerShell is a bit cumbersome, I prefer to pin shortcuts to the Windows PowerShell console (and the Windows PowerShell ISE) to both the Start window and the Windows taskbar. This technique of pinning shortcuts to the applications, as shown in Figure 1-2, provides single-click access to Windows PowerShell from wherever I might be working.

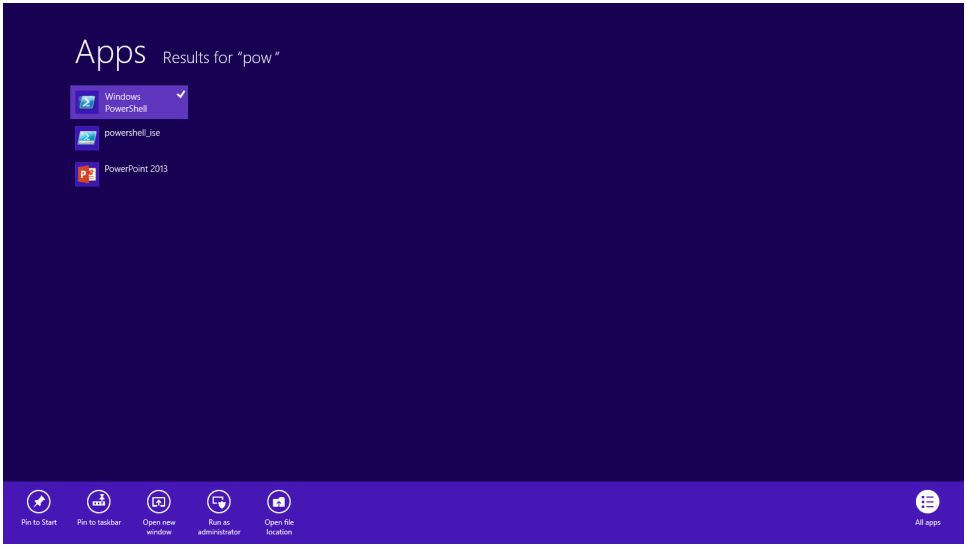


FIGURE 1-2 By right-clicking the Windows PowerShell icon in the Search results box, the Pin to Start and the Pin to taskbar options appear.

On Windows Server 2012, it is unnecessary to find the icon by using the Search box on the Start window because an icon for the Windows PowerShell console exists by default on the taskbar of the desktop.

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

Security issues with Windows PowerShell

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

Running as a non-elevated user

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

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

```
PS C:\> get-disk
get-disk : Access to a CIM resource was not available to the client.
At line:1 char:1
+ get-disk
+ ~~~~~~~~
    + CategoryInfo          : PermissionDenied: (MSFT_Disk:ROOT/Microsoft/Windows/S
   torage/MSFT_Disk) [Get-Disk], CimException
    + FullyQualifiedErrorId : MI RESULT 2,Get-Disk
```

TIP **If you attempt to run cmdlets that require elevated rights, you will encounter inconsistencies with errors. For example, in a non-elevated Windows PowerShell console, the error from the *Get-Disk* cmdlet is Access To A CIM Resource Was Not Available To The Client. The error from the *Stop-Service* cmdlet is Cannot Open XXX Service On Computer. The *Get-VM* cmdlet simply returns no information and no error. Therefore, check for console rights as a first step in troubleshooting.**

Launching Windows PowerShell with administrator rights

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

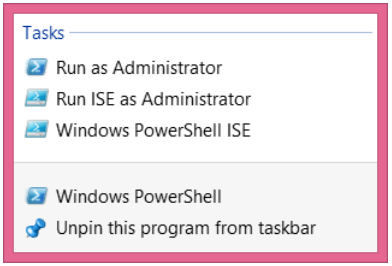


FIGURE 1-3 Right-click the Windows PowerShell icon to bring up the option to Run as Administrator.

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

NOTE **It is possible to avoid this prompt by turning off UAC. However, UAC is a significant security feature, so I do not recommend disabling it. The UAC has been fine-tuned on Windows 7 and Windows 8. The number of UAC prompts has been greatly reduced from the number that used to exist with the introduction of UAC on Windows Vista.**

Now that you are running Windows PowerShell with administrator rights, you can do anything your account has permission to do. For example, if you run the *Get-Disk* cmdlets, you will see information similar to the following:

```
PS C:\> get-disk

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

Using Windows PowerShell cmdlets

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

The most common verb: *Get*

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

Once the Windows PowerShell console appears, run the *Get-Process* cmdlet. To do this, use the Windows PowerShell Tab Completion feature to complete the cmdlet name. Once the cmdlet name appears, press the Enter key to cause the command to execute.

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

Finding process information

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

```
Get-Pro
```

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

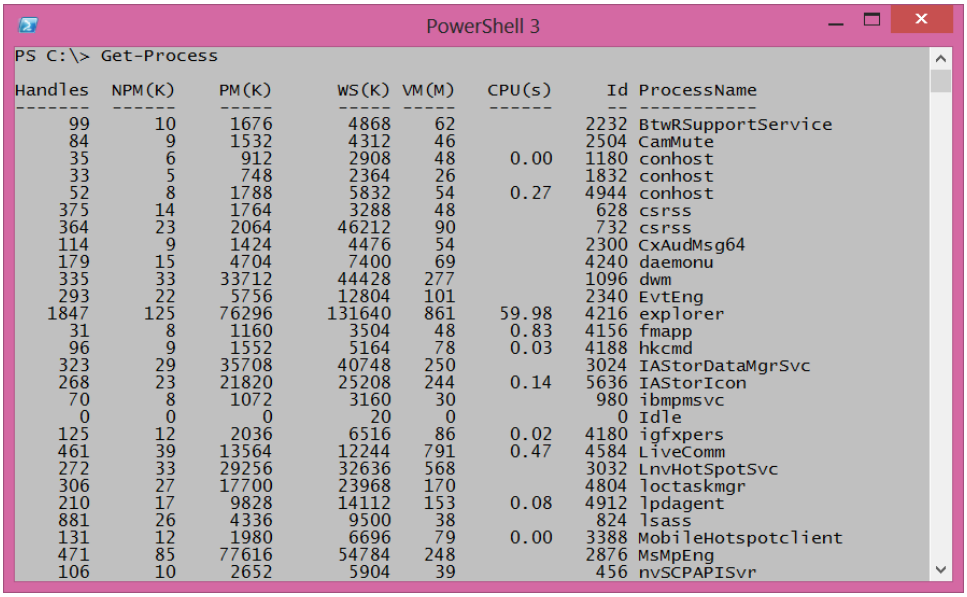


FIGURE 1-4 The Windows PowerShell *Get-Process* cmdlet returns detailed Windows process information.

To find information about Windows services, use the verb *Get* and the noun *Service*. In the Windows PowerShell console, type the following, then press the Tab key followed by Enter:

```
Get-Servi
```

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

Identifying installed Windows hotfixes

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

```
Get-Hotf
```

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

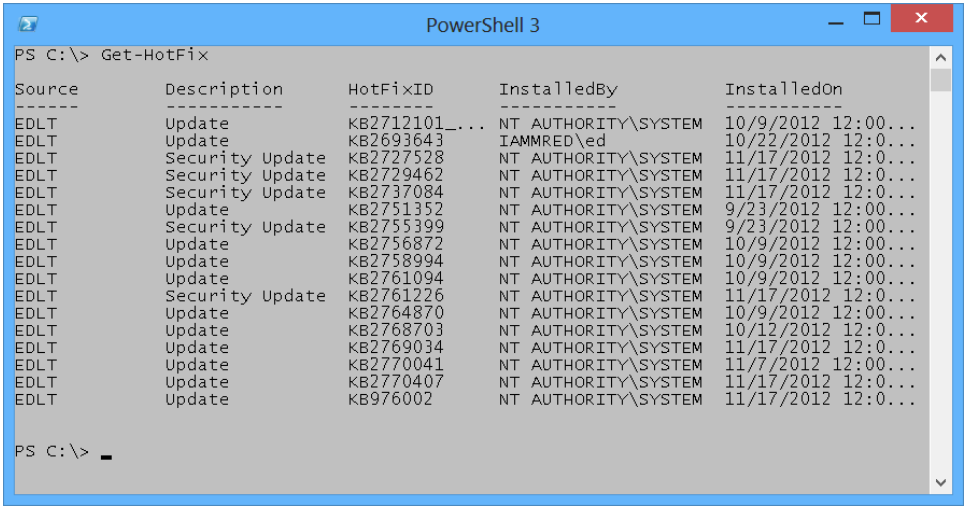


FIGURE 1-5 Use the *Get-Hotfix* cmdlet to obtain a detailed listing of all applied Windows hotfixes.

Getting detailed service information

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

```
Get-Servi
```

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

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

```
PS C:\> Get-Service

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

Identifying installed network adapters

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

```
Get-NetA
```

The following example shows the command and associated output:

```
PS C:\> Get-NetAdapter

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

Retrieving detected network connection profiles

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

```
Get-NetC
```

The following example shows the command and associated output:

```
PS C:\> Get-NetConnectionProfile


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

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

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

Getting the current culture settings

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

```
Get-Cu
```

When the command runs basic information such as the Language Code ID number (LCID), the name of the culture settings, in addition to the display name of the culture settings,

return to the Windows PowerShell console. The following example shows the command and associated output:

```
PS C:\> Get-Culture

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

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

```
Get-Ui
```

The following example shows the command and associated output:

```
PS C:\> Get-UICulture

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

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

Finding the current date and time

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

```
Get-Dat
```

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

```
Get-Date
```

The following example shows the command and associated output:

```
PS C:\> Get-Date

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

Generating a random number

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

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

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

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

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

```
Get-R
```

The following example shows the command and associated output:

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

Supplying options for cmdlets

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

Using single parameters

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

Finding specific types of hotfixes

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

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

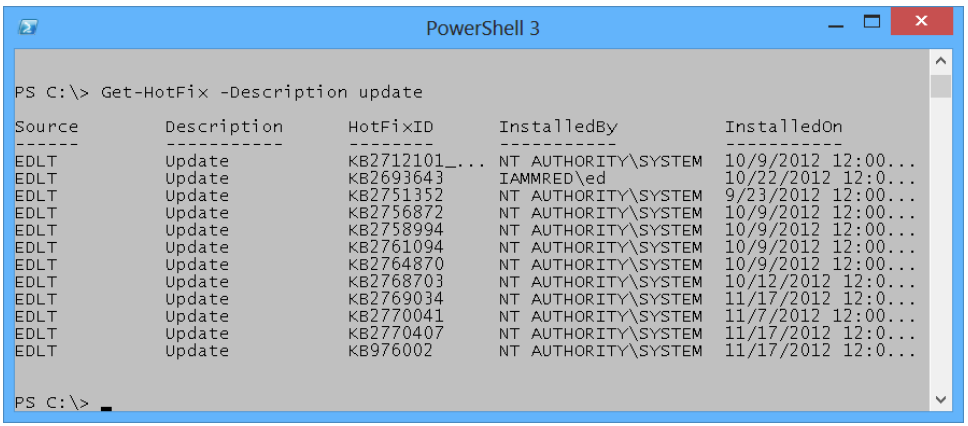


FIGURE 1-6 Add the -Description parameter to the *Get-HotFix* cmdlet to see specific hotfixes such as updates in a filtered list.

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

```
PS C:\> Get-HotFix update
Get-HotFix : Cannot find the requested hotfix on the 'localhost' computer. Verify
the input and run the command again.
At line:1 char:1
+ Get-HotFix update
+ ~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (:) [Get-HotFix], ArgumentException
    + FullyQualifiedErrorId : GetHotFixNoEntriesFound,Microsoft.PowerShell.Commands
   .GetHotFixCommand
```

The error, while not really clear, seems to indicate that the *Get-HotFix* cmdlet attempts to find a hotfix named update. This is, in fact, the attempted behavior. The Help file information for the *Get-HotFix* cmdlet reveals that *-ID* is position 1, as shown in the following example:

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

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

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

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

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

Finding specific processes

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

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

The following example shows the command and associated output:

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

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    607      39   144552     164652   718     5.58   4860 powershell
```

You can tell the *Get-Process* cmdlet accepts the -Name parameter in a positional manner because the Help file states it is in position 1. The following example shows this position:

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

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

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

Generating random numbers in a range

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

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

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

```
Get-R + <TAB> + <SPACE> + 21
```

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

```
PS C:\> Get-Random 1 21
Get-Random : A positional parameter cannot be found that accepts argument '21'.
At line:1 char:1
+ Get-Random 1 21
+ ~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-Random], ParameterBindingEx
   ception
    + FullyQualifiedErrorId : PositionalParameterNotFound,Microsoft.PowerShell.Comm
   ands.GetRandomCommand
```

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

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

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

The following example shows the command and associated output:

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

Introduction to parameter sets

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

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

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

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

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

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

Generating a certain number of random numbers

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

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

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

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

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

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

```
Get-Random -InputObject (1..10) -Count 5
```

The following example shows the command and associated output:

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

Using command-line utilities

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

```
PS C:\> ipconfig

Windows IP Configuration

Wireless LAN adapter Local Area Connection* 14:

   Media State . . . . . . . . . . . : Media disconnected

   Connection-specific DNS Suffix  . :

Ethernet adapter vEthernet (WirelessSwitch):

   Connection-specific DNS Suffix  . : quadriga.com

   Link-local IPv6 Address . . . . . : fe80::915e:d324:aa0f:a54b%31

   IPv4 Address. . . . . . . . . . . : 192.168.13.220

   Subnet Mask . . . . . . . . . . . : 255.255.248.0

   Default Gateway . . . . . . . . . : 192.168.15.254

Wireless LAN adapter Local Area Connection* 12:

   Media State . . . . . . . . . . . : Media disconnected

   Connection-specific DNS Suffix  . :

Ethernet adapter vEthernet (InternalSwitch):

   Connection-specific DNS Suffix  . :

   Link-local IPv6 Address . . . . . : fe80::bd2d:5283:5572:5e77%19

   IPv4 Address. . . . . . . . . . . : 192.168.3.228
```

```
   Subnet Mask . . . . . . . . . . . : 255.255.255.0

   Default Gateway . . . . . . . . . : 192.168.3.100

<OUTPUT TRUNCATED>
```

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

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

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

Working with Help options

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

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

```
Update-Help -Force
```

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

To display Help information from the Internet, use the *-Online* switch. When used in this way, Windows PowerShell causes the default browser to open to the appropriate page from the Microsoft TechNet website.

In an enterprise, network administrators might want to use the *Save-Help* cmdlet to download Help from the Internet. Once downloaded, the *Update-Help* cmdlet can point to the network share for the files. This is an easy task to automate and can run as a scheduled task.

Summary

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

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

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

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

The first thing you should do when logging onto the Windows PowerShell console is to run the *Update-Help* cmdlet. Note that this requires administrator rights and an Internet connection.

CHAPTER 2

Using Windows PowerShell cmdlets

- Understanding the basics of cmdlets
- Searching the Help topics
- Using the *Get-Command* to find cmdlets
- Using the *Get-Member* cmdlet
- Using the *Show-Command* cmdlet
- Setting the Script Execution Policy
- Creating a basic Windows PowerShell profile

Once you have an understanding of Windows PowerShell cmdlet naming conventions and how to use Windows PowerShell cmdlets, it is time to expand upon that knowledge by looking at the use of Windows PowerShell common parameters. One of the great things about Windows PowerShell is the way it is both intuitive and self-describing. Once you begin to get a feel for the way that Windows PowerShell cmdlets work, you will be able to sense (or feel) how Windows PowerShell cmdlets should behave. This intuitive design of Windows PowerShell is integral, and permits you to leverage knowledge of Windows PowerShell across multiple platforms. For example, if you learn how to use Windows PowerShell to manage a Windows 8 computer, you will be able to figure out how to use Windows Server 2012 cmdlets because Windows PowerShell cmdlets should always behave in the same manner.

To assist you in your quest to learn Windows PowerShell, you will need four basic tools: *Get-Help*, *Get-Command*, *Get-Member,* and *Show-Command*. These four Windows PowerShell cmdlets form the foundation upon which you will build your tower of Windows PowerShell knowledge. As you are beginning to work more and more with Windows PowerShell, you will want to customize the Windows PowerShell console. To do this, you will need to first modify the Windows PowerShell Script Execution Policy. This is likely to be your first major change to the Windows PowerShell defaults and is not something to take lightly.

Understanding the basics of cmdlets

All Windows PowerShell cmdlets behave basically the same way. There are some idiosyncrasies between cmdlets from different vendors or different teams at Microsoft, but in general, once you understand the way that Windows PowerShell cmdlets work, you can transfer the knowledge to other cmdlets, platforms, and applications. To call a Windows PowerShell cmdlet, type it on a line at the command prompt in the Windows PowerShell console. To modify the way the cmdlet retrieves or displays information, supply options for parameters that modify the cmdlet. Many of these parameters are unique and apply only to certain cmdlets. However, some parameters are applicable to all Windows PowerShell cmdlets. In fact, these cmdlets are part of the strength of the Windows PowerShell design. Called *common parameters*, the parameters supported by all Windows PowerShell cmdlets appear in the next section.

Common Windows PowerShell parameters

All Windows PowerShell cmdlets support common parameters. The following list shows the common parameters. Each of the common parameters also permits the use of an alias for the parameter. The alias for each parameter appears in parentheses:

- Verbose (vb)
- Debug (db)
- WarningAction (wa)
- WarningVariable (wv)
- ErrorAction (ea)
- ErrorVariable (ev)
- OutVariable (ov)
- OutBuffer (ob)

If a Windows PowerShell cmdlet changes system state (such as stopping a process or changing the start-up value of a service), two additional parameters become available:

- WhatIf (wi)
- Confirm (cf)

Using the Verbose parameter to provide additional information

As an example of how to use common parameters in Windows PowerShell, you can use the -Verbose parameter to obtain additional information about the action a cmdlet performs.

The following command stops all instances of the Notepad.exe process running on the local system (there is no output from the command):

```
PS C:\> Stop-Process -Name notepad
PS C:\>
```

To see what processes stop in response to the *Stop-Process* cmdlet, use the -Verbose common parameter. In the following example, two separate Notepad.exe processes stop in response to the *Stop-Process* cmdlet. Because the cmdlet uses the -Verbose common parameter, detailed information about each process appears in the output:

```
PS C:\> Stop-Process -Name notepad -Verbose
VERBOSE: Performing operation "Stop-Process" on Target "notepad (5564)".
VERBOSE: Performing operation "Stop-Process" on Target "notepad (5924)".
PS C:\>
```

Using the ErrorAction parameter to hide errors

When you use the *Stop-Process* cmdlet to stop a process, a nasty error displays on the Windows PowerShell console if there is no instance of the specified process running. In the following example, the *Stop-Process* cmdlet attempts to stop a process named Notepad.exe, but there are no instances of the Notepad.exe process running. Therefore, the following error displays:

```
PS C:\> Get-Process -Name notepad
Get-Process : Cannot find a process with the name "notepad". Verify the process
name and call the cmdlet again.
At line:1 char:1
+ Get-Process -Name notepad
+ ~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (notepad:String) [Get-Process], Proce
   ssCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Comma
   nds.GetProcessCommand

PS C:\>
```

If you know, or at least suspect, that a process is not running, but you would like to verify this, you might want to use the -ErrorAction common parameter. To hide error messages arising from the *Get-Process* cmdlet, supply a value of SilentlyContinue for the -ErrorAction parameter prior to running the cmdlet. The following example shows this technique:

```
PS C:\> Get-Process -Name notepad -ErrorAction SilentlyContinue
PS C:\>
```

> ***NOTE*** **The preceding command appears to be really long, but keep in mind that tab expansion makes this easy to type correctly. In fact, the previous command is *Get-Pro<tab><space>-n<tab><space>notepad<space>-e<tab><space>s<tab>*.**

You can use the parameter alias *-EA* instead of typing -ErrorAction (although with tab expansion it is exactly the same number of keystrokes, whether *-E<tab>* or *-EA*) to shorten the command. In addition, when you work with the *Get-Process* cmdlet, the default parameter set is *Name*. This means that the -Name parameter from *Get-Process* is the default parameter,

and therefore *Get-Process* interprets any string in the first position as the name of a process. The following example shows the revised command:

```
PS C:\> Get-Process notepad -ea SilentlyContinue

PS C:\>
```

If you are uncertain of valid values for the -ErrorAction parameter, you can supply anything to the parameter and then carefully read the resulting error message. In the text of the error message, the first two lines state that Windows PowerShell is unable to convert the value to the *System.Management.Automation.ActionPreference* type. The fourth line of the error message lists allowed values for the -ErrorAction parameter. The allowed values are *SilentlyContinue*, *Stop*, *Continue*, *Inquire*, and *Ignore*. Figure 2-1 shows this technique of forcing an error.

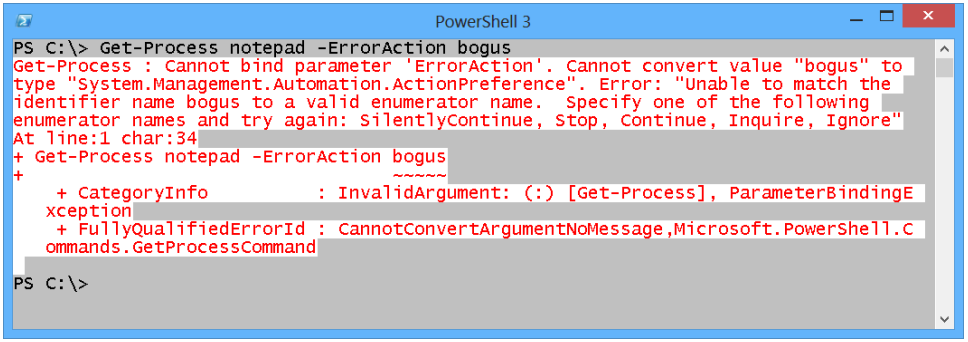


FIGURE 2-1 Forcing an intentional error reveals permissible values for Windows PowerShell parameters.

Starting the Windows PowerShell transcript

One of the great features of the Windows PowerShell console is the *Start-Transcript* cmdlet. To start a Windows PowerShell transcript, type *Start-Transcript* on a blank line in the Windows PowerShell console. The following example shows the command and associated output:

```
PS C:\> Start-Transcript
Transcript started, output file is C:\Users\ed.IAMMRED\Documents\PowerShell_transcri
pt.20121120142251.txt
```

Once you start the Windows PowerShell transcript, all commands, command output, and even error messages appear in the transcript file. The transcript file is useful for several reasons. The following list shows some of these reasons:

- **As a troubleshooting tool** The transcript becomes a valuable troubleshooting tool showing commands and specific error messages arising from the commands.
- **As a learning tool** When you explore a variety of Windows PowerShell commands and you find useful commands, you already have a record of the commands and associated output from the commands.

- **As an audit tool** The transcript provides the user name, the time, and a record of all commands and output from those commands.

Stopping and reviewing the Windows PowerShell transcript

The Windows PowerShell transcript contains all commands and output from the commands, including errors. To view the Windows PowerShell transcript, you must stop the transcript first. To do this, use the *Stop-Transcript* cmdlet. Once the transcript stops, the path to the transcript file appears in the Windows PowerShell console, as shown in the following example:

```
PS C:\> Stop-Transcript

Transcript stopped, output file is C:\Users\ed.IAMMRED\Documents\PowerShell_transcri
pt.20121122074731.txt

PS C:\>
```

You can use Windows PowerShell to parse the transcript file (it is plain text) or you can open it in Notepad. Figure 2-2 shows the Windows PowerShell transcript log file in Notepad.

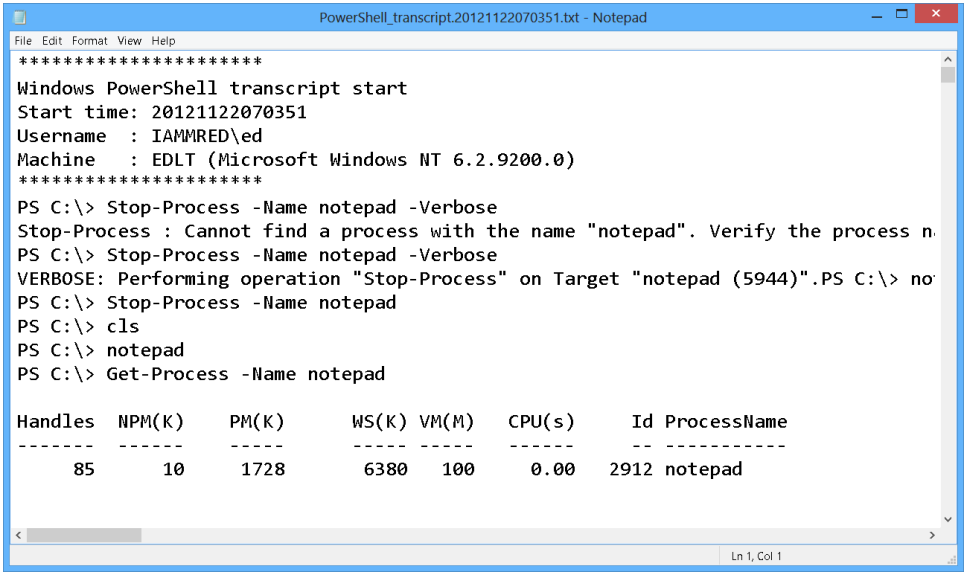


FIGURE 2-2 The Windows PowerShell transcript file opens easily in Notepad and shows both commands and errors.

Searching the Help topics

There are two types of Help topics available from within Windows PowerShell. The first type of topic describes cmdlets and how to use the cmdlets. The cmdlet Help topics typically contain a basic description of the cmdlet, the syntax, an explanation of the parameters, and examples of using the cmdlet. The examples typically range from extremely simple to moderately complex examples involving more than one cmdlet. You access all the cmdlet Help topics through the *Get-Help* cmdlet (or the *Help* function) when you supply a cmdlet name.

The second type of Help topic is the conceptual topic. The conceptual Help topics do not contain multiple sections (such as description, syntax, or examples); instead, they are single text files. These files do not describe single cmdlets, but rather provide detailed explanations of fundamental Windows PowerShell concepts. For example, there are Help files that cover the WMI Query Language (WQL), Windows PowerShell remoting, workflow, and even error handling. You access all the conceptual topics through the *Get-Help* cmdlet (or the *Help* function) when you supply a phrase beginning with "About_". You can use the *Get-Help* cmdlet and tab expansion to cycle through the conceptual Help topics. To do this, perform the following at the command prompt:

```
Get-Help About_ + <TAB> + <ENTER>
```

On my Windows 8 computer, there are currently 110 About_ conceptual topics comprising 47,190 words of information.

> ***NOTE*** **Before you use Windows PowerShell Help, you must run the *Update-Help* cmdlet to download the latest Help files. This command requires administrator rights (see Chapter 1, "Overview of Windows PowerShell 3.0") and an Internet connection to run.**

Using the *Get-Help* cmdlet

To find Help information about using a specific Windows PowerShell cmdlet, use the *Get-Help* cmdlet and supply the name of the cmdlet to the -Name parameter. When the *Get-Help* cmdlet runs in this mode, it provides basic Help information, including the following elements: Name, Synopsis, Syntax, Description, Related Links, and Remarks. The following example shows this type of command:

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

The default parameter for the *Get-Help* cmdlet is the -Name parameter, and therefore it is not necessary to specify the -Name parameter each time you want to find cmdlet Help. For example, the following command also displays Help information about the *Get-Process* cmdlet:

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

After you are familiar with the basics of how a particular cmdlet works, you will need only a quick reminder of how to use the cmdlet. On these occasions, using the *-Examples* switch does the trick. When run with the -Examples parameter, the *Get-Help* cmdlet returns the following elements: Name, Synopsis, and the Examples portion from the Help topic. The following example shows this command:

```
Get-help -name Get-Service -Examples
```

Once you begin to explore some of the more powerful features of Windows PowerShell cmdlets, or you need complete Windows PowerShell cmdlet information, use the *-Full* switch. The *-Full* switch forces Windows PowerShell to return all available Help information. This information includes the following elements: Name, Synopsis, Syntax, Description, Parameters, Inputs, Outputs, Notes, Examples, Related Links, and Remarks. In addition, the Parameters section includes information as to whether or not a parameter is required, its position, default values, and if it accepts pipelined input or wildcards.

Paging the Help output

When you use the *Get-Help* cmdlet to display all the cmdlet Help information by using the -Full parameter, you will get several pages of information. By default, the Help information scrolls to the Windows PowerShell console. As long as the amount of information does not exceed the console buffer, you can scroll back to the beginning of the Help output and read through the data. But if the information exceeds the console buffer, you will need to either change the buffer settings or send the output to a pager. To cause the output to display a single page of information and halt until you press either the spacebar or Enter, use the *Help* function. The *Help* function works similarly to the *Get-Help* cmdlet because it accepts the same parameters and switches. The difference is that instead of dumping all the information directly to the Windows PowerShell console, it sends the data through a pager that displays one page of information at a time. This pager is sophisticated and detects the size of the Windows PowerShell console prior to figuring out the amount of information to display at a time. Therefore, if you have a small Windows PowerShell console window, then only a few lines at a time display. If you have a large Windows PowerShell console window, it displays more lines of output. The following command displays complete cmdlet Help information one page at a time for the *Get-WinEvent* cmdlet:

```
Help -Full -Name Get-WinEvent
```

Figure 2-3 illustrates using the *Help* function to display paged information for the *Get-WinEvent* cmdlet.

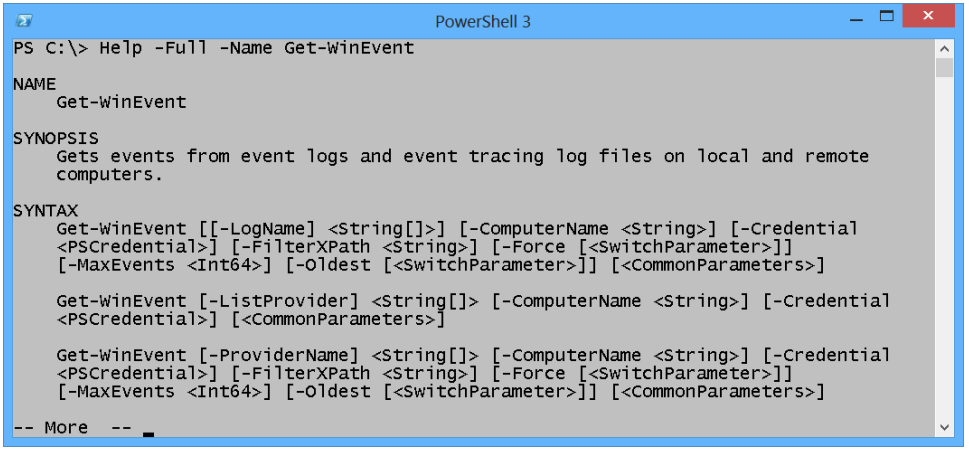


FIGURE 2-3 Use the *Help* function to display complete cmdlet Help information one page at a time.

Using the *Get-Help* cmdlet to search for Windows PowerShell cmdlets

Because the *Get-Help* cmdlet accepts wildcard characters, you can use *Get-Help* to find cmdlets related to a specific topic. For example, to use *Get-Help* to find cmdlets related to processes, use a command such as the one in the following example:

```
Get-Help *process
```

The benefit is that because you use wildcard characters you can target specific types of cmdlets with minimal typing. The following example shows this technique:

```
PS C:\> get-help *P*ce?s

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
Debug-Process                     Cmdlet    Microsoft.PowerShell.M... Debugs one ...
Get-Process                       Cmdlet    Microsoft.PowerShell.M... Gets the pr...
Start-Process                     Cmdlet    Microsoft.PowerShell.M... Starts one ...
Stop-Process                      Cmdlet    Microsoft.PowerShell.M... Stops one o...
Wait-Process                      Cmdlet    Microsoft.PowerShell.M... Waits for t...
```

If you are specifically interested in cmdlet Help, specify the cmdlet category when you use the *Get-Help* cmdlet, as shown in the following example:

```
Get-Help process -Category cmdlet
```

Using the About conceptual Help topics

It is possible to use the *Get-Help* cmdlet and the -Name parameter to search for About_ conceptual Help topics. But using wildcards is extra work and can also lead to missing topics. Following is an example of using wildcards:

```
PS C:\> Get-Help -Name about*wmi*

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
about_WMI                         HelpFile
about_WMI_Cmdlets                 HelpFile                            Provides ba...
```

Without the trailing wildcard character, the command returns only the about_WMI topic. A better way of searching the About_ conceptual Help topics is to specify the HelpFile *-Category*. This makes for a cleaner command and is actually easier to type. The following example shows the command and associated output:

```
PS C:\> Get-Help -Name wmi -Category HelpFile

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
about_WMI                         HelpFile
about_WMI_Cmdlets                 HelpFile                            Provides ba...
```

> ***NOTE*** **The preceding command appears to be a longer command. However, when you use tab expansion for cmdlet name, parameter name, and the permitted values for the parameters, the command actually takes fewer keystrokes than the one using wildcard characters.**

To find conceptual Help topics related to common parameters, use the *Get-Help* cmdlet and specify the word common while choosing the *HelpFile* category, as shown in the following example:

```
PS C:\> Get-Help common -Category HelpFile

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
about_CommonParameters            HelpFile                            Describes t...
about_ActivityCommonParameters    HelpFile                            Describes t...
about_WorkflowCommonParameters    HelpFile                            This topic ...
```

Because some of the conceptual Help topics are rather long, you might want to use the *Help* function instead of using the *Get-Help* cmdlet. You can use the same technique. For example, to find conceptual Help files related to operators, you need to type only a portion of the keyword operator while supplying the *HelpFile* value to the -Category parameter, as shown in the following example:

```
PS C:\> Help oper -Category HelpFile

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
about_Arithmetic_Operators        HelpFile                            Describes t...
about_Assignment_Operators        HelpFile                            Describes h...
about_Comparison_Operators        HelpFile                            Describes t...
about_Logical_Operators           HelpFile                            Describes t...
about_Operators                   HelpFile                            Describes t...
about_Operator_Precedence         HelpFile                            Lists the W...
about_Properties                  HelpFile                            Describes h...
about_Type_Operators              HelpFile                            Describes t...
```

Using the *Get-Command* to find cmdlets

There are several different ways to use the *Get-Command* cmdlet to find Windows PowerShell cmdlets. The most basic way to use the *Get-Command* cmdlet is to use it to find cmdlets that use a verb such as Get, as shown in the following example:

```
Get-Command -Verb Get
```

There are 98 authorized verbs in Windows PowerShell 3.0. The listing of permissible verbs appears when you use the *Get-Verb* function. Figure 2-4 shows the use of the *Get-Verb* function as well as a portion of the associated output.

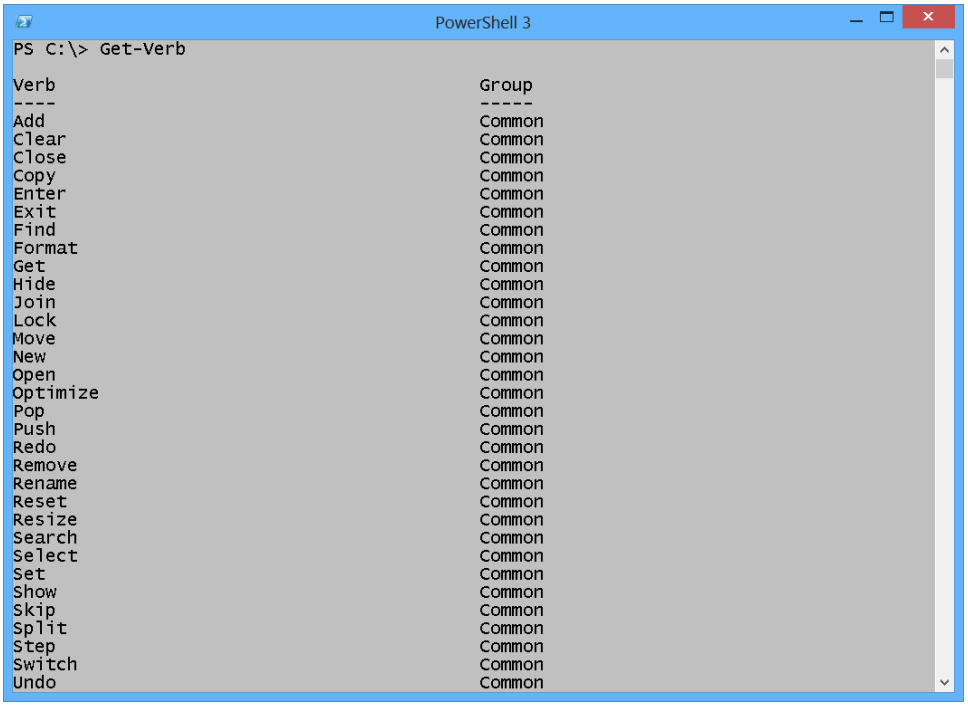


FIGURE 2-4 Use the *Get-Verb* cmdlet to display the listing of approved Windows PowerShell verbs.

If you want to retrieve, gather, or collect information, you probably should use the Get verb. To do this, use the *Get-Command* cmdlet while supplying the value get to the -Verb parameter, as shown in the following example:

```
Get-Command -Verb get
```

On my Windows 8 computer with the Windows Server 2012 Remote Server Administration Tools (RSAT) installed, I have 577 cmdlets (and functions) returned that use the verb Get. Because of the large number of Get cmdlets (and functions), you probably should also use the -Noun parameter to reduce the number of items to sort through. Because most of the Windows PowerShell cmdlet names are related to the feature of technology they manage, it is possible to use the -Noun parameter to aid in cmdlet discovery. The following example shows how to use the -Noun parameter to find cmdlets that contain the letters TCP:

```
PS C:\> Get-Command -Verb get -Noun *tcp*

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Get-NetTCPConnection                               NetTCPIP
Function        Get-NetTCPSetting                                  NetTCPIP


PS C:\>
```

If you are interested in cmdlets related to IP, however, the results from a wildcard search will be somewhat disappointing. This is because the letter pattern *ip* appears in many cmdlet (and function) names. Figure 2-5 shows the command and associated results.

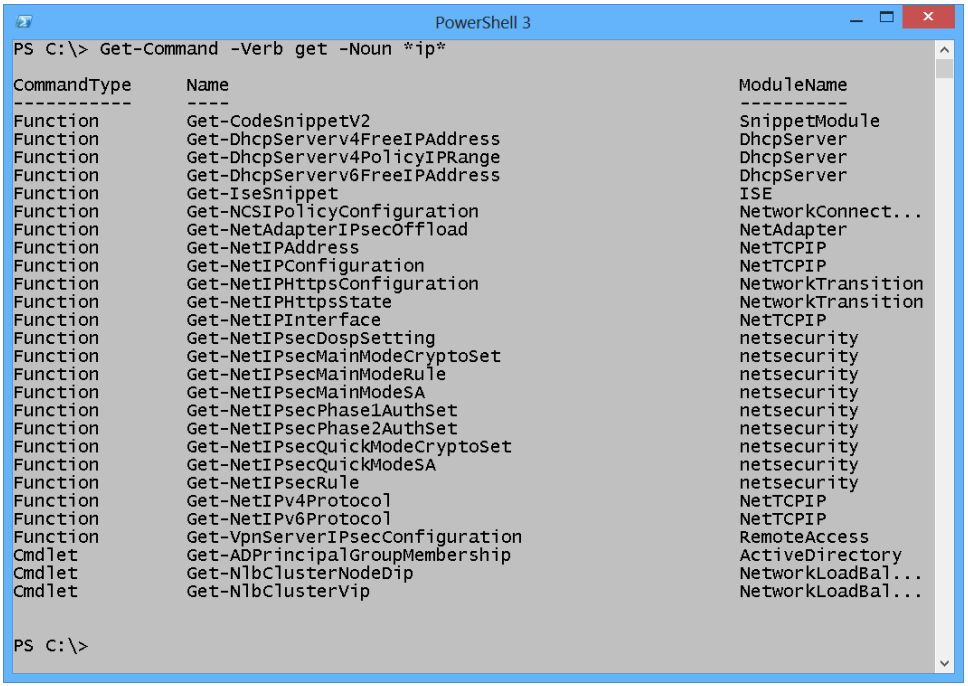


FIGURE 2-5 When a short letter pattern appears in the cmdlet name, wildcard patterns are ineffective.

When you want to find cmdlets related specifically to a certain technology, use the -Module parameter. This is because Windows PowerShell cmdlets (and functions) reside in modules, and the modules group management functionality around specific technology. Therefore, to accomplish TCP/IP management, use cmdlets from the *NetTCPIP* module, as shown in the following example:

```
PS C:\> Get-Command -Verb get -Noun *ip* -Module NetTcpIp

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Get-NetIPAddress                                   NetTCPIP
Function        Get-NetIPConfiguration                             NetTCPIP
Function        Get-NetIPInterface                                 NetTCPIP
Function        Get-NetIPv4Protocol                                NetTCPIP
Function        Get-NetIPv6Protocol                                NetTCPIP
```

To find all the cmdlets (functions) contained in the *NetTCPIP* module, use the *w* cmdlet and specify only the -Module parameter. In this way, all nouns and all verbs appear in the output. The following example shows the command:

```
Get-Command -Module NetTcpIp
```

Keep in mind you can use tab expansion to complete the module name. You can also use wildcard characters.

If you are specifically interested in configuring objects, typically the verb to use is the Set verb. The following example shows the Windows PowerShell functions using the Set verb from the *NetTCPIP* module:

```
PS C:\> Get-Command -Module NetTCPIP -Verb set

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Set-NetIPAddress                                   NetTCPIP
Function        Set-NetIPInterface                                 NetTCPIP
Function        Set-NetIPv4Protocol                                NetTCPIP
Function        Set-NetIPv6Protocol                                NetTCPIP
Function        Set-NetNeighbor                                    NetTCPIP
Function        Set-NetOffloadGlobalSetting                        NetTCPIP
Function        Set-NetRoute                                       NetTCPIP
Function        Set-NetTCPSetting                                  NetTCPIP
Function        Set-NetUDPSetting                                  NetTCPIP
```

Using the *Get-Member* cmdlet

To find the properties, methods, or events of an object in Windows PowerShell, use the *Get-Member* cmdlet. Properties of an object describe the object. For example, an automobile has properties such as color, body style, miles per gallon (MPG), and cost. Methods perform actions. For example, an automobile has the "drive down the road" method and the "stop at a stop sign" method, and probably the "play music" method as well.

Everything in Windows PowerShell is an object. In fact, one of the characteristics that makes Windows PowerShell unique among shell and scripting environments is that it returns objects and pipelines objects. Most other shells and scripting environments are text-based. With text-based shells, you have to use a separate tool to parse the text to find the information. This is one problem with running old commands (such as *ping*, *ipconfig*, and *tracert*) inside Windows PowerShell: The commands return text, and picking out a specific piece of information is difficult.

There are many different ways of using the *Get-Member* cmdlet. The cmdlet has parameters for filtering out results and for searching for specific parameters. To see all the different ways to use the *Get-Member* cmdlet, use the *Get-Help* cmdlet discussed earlier in this chapter. The following example shows how the command returns basic information about using the *Get-Member* cmdlet:

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

To see only the examples of using the *Get-Member* cmdlet, use the command in the following example:

```
Get-Help Get-Member -Examples
```

To see the complete Help information, use the following command:

```
Get-Help Get-Member -Full
```

Exploring property members

To explore the property members of an object, use the *Get-Member* cmdlet and use the -MemberType parameter to choose the member type of Property. In addition, you need to supply a value for the InputObject parameter. The InputObject parameter accepts the object whose property members you want to display. To ensure the object is created prior to the displaying property members, you must use a pair of parentheses and place the cmdlet producing the objects inside.

The following command creates the *System.Datetime* object by executing the *Get-Date* cmdlet. It then uses the *Get-Member* cmdlet to display property members:

```
PS C:\> Get-Member -InputObject (get-date) -MemberType Property

   TypeName: System.DateTime

Name        MemberType Definition
----        ---------- ----------
Date        Property   datetime Date {get;}
Day         Property   int Day {get;}
DayOfWeek   Property   System.DayOfWeek DayOfWeek {get;}
DayOfYear   Property   int DayOfYear {get;}
Hour        Property   int Hour {get;}
Kind        Property   System.DateTimeKind Kind {get;}
Millisecond Property   int Millisecond {get;}
Minute      Property   int Minute {get;}
Month       Property   int Month {get;}
Second      Property   int Second {get;}
Ticks       Property   long Ticks {get;}
TimeOfDay   Property   timespan TimeOfDay {get;}
Year        Property   int Year {get;}
```

> ***NOTE*** **There is a difference in the *–MemberType* of *Property* and the *–MemberType* of *Properties*. The former only lists properties of an object. The latter lists properties of an object, but additionally lists *ScriptProperty*, *NoteProperty*, *CodeProperty,* and others.**

Using the *Show-Command* cmdlet

The *Show-Command* cmdlet displays a graphical input control for the cmdlet specified to the -Name parameter. The following command displays the graphical input control for the *Get-Process* cmdlet:

```
Show-Command -Name Get-Process
```

Because the -Name parameter is the default parameter for the *Show-Command* cmdlet, you do not need to always specify the -Name parameter. Therefore, the following command, which omits the -Name parameter, also works:

```
Show-Command Get-Process
```

Once the command runs, it displays a graphical input control. Each control is specific to the cmdlet specified at run time. The input control for the *Get-Process* cmdlet shows the three different parameter sets, one on each tab. The default parameter set (Name) appears on the first tab. By using the *Show-Command* cmdlet, you can create a command by using the mouse or the keyboard to select and supply the options for a specific command. Once created, you have the option to copy the command to the Clipboard or run the command. When you choose to run the command, the complete Windows PowerShell command first displays to the Windows PowerShell console command line. Next, it executes and displays the results on subsequent lines. Figure 2-6 shows the input control for the *Get-Process* cmdlet.

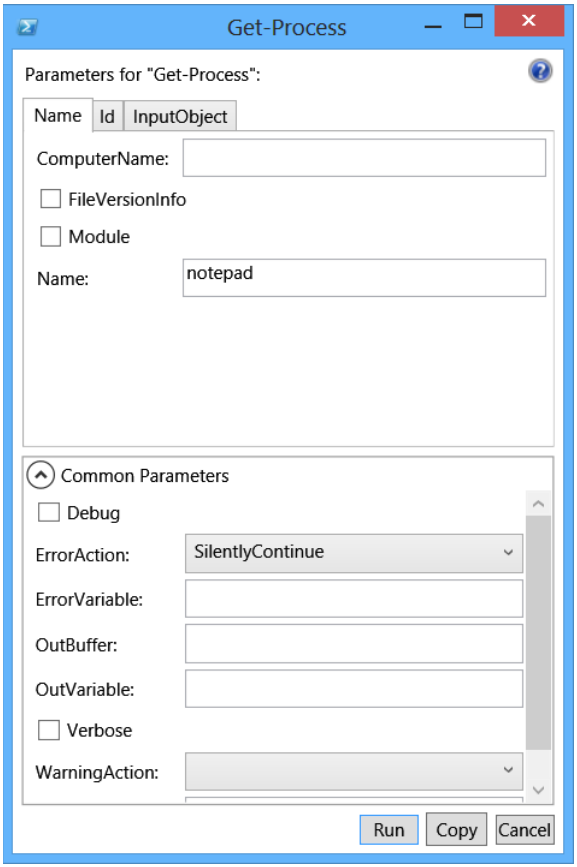


FIGURE 2-6 The graphical control for the *Get-Process* cmdlet created by the *Show-Command* cmdlet.

Setting the Script Execution Policy

By default, Windows PowerShell disallows the execution of scripts. Typically, Group Policy controls script support. If it does not, and if you have administrator rights on your computer, you can use the Windows PowerShell *Set-ExecutionPolicy* cmdlet to turn on script support. You can enable one of six levels by using the *Set-ExecutionPolicy* cmdlet. (The discussion of Windows PowerShell scripts begins in Chapter 10). The following list shows the options:

- **Restricted** Does not load configuration files or run scripts. Restricted is the default.
- **AllSigned** Requires that a trusted publisher sign all scripts and configuration files, including scripts that you write on the local computer.
- **RemoteSigned** Requires that a trusted publisher sign all scripts and configuration files downloaded from the Internet.
- **Unrestricted** Loads all configuration files and runs all scripts. If you run an unsigned script that was downloaded from the Internet, you are prompted for permission before it runs.
- **Bypass** Nothing is blocked and there are no warnings or prompts.
- **Undefined** Removes the currently assigned execution policy from the current scope. This parameter does not remove an execution policy that is set in a Group Policy scope.

NOTE **If the script execution policy is set through Group Policy, you cannot change it, even with administrator rights on the local machine.**

In addition to six levels of execution policy, there are three different scopes for the execution policies:

- **Process** The execution policy affects only the current PowerShell process.
- **CurrentUser** The execution policy affects only the current user.
- **LocalMachine** The execution policy affects all users of the computer.

To set the *LocalMachine* execution policy, you must have administrator rights on the local computer. The following command shows a local administrator changing the script execution policy to *unrestricted*:

```
Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy unrestricted
```

By default, non-elevated users have rights to set the script execution policy for the *CurrentUser* user scope that affects their own execution policy. The following command shows a non-elevated user setting the script execution policy to *remotesigned*:

```
PS C:\> Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy remotesigned
```

```
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing
the execution policy might expose you to the security risks described in the
about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution
policy?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
```

With so many choices available to you for a script execution policy, you might wonder which one is appropriate for you. The Windows PowerShell team recommends the *RemoteSigned* setting, stating that it is appropriate for most circumstances. Remember that even though descriptions of the various policy settings use the term *Internet*, this might not always refer to the World Wide Web or even to locations outside your own firewall. This is because Windows PowerShell obtains its script origin information by using the Windows Internet Explorer zone settings. This means anything that comes from a computer other than your own is in the Internet zone. You can change the Internet Explorer zone settings by using Internet Explorer, the registry, or Group Policy.

If you do not want to see the confirmation message when you change the script execution policy in Windows PowerShell 3.0, use the -Force parameter.

To view the execution policy for all scopes, use the list parameter when calling the *Get-ExecutionPolicy* cmdlet, as shown in the following example:

```
PS C:\> Get-ExecutionPolicy -List

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser    RemoteSigned
 LocalMachine      Restricted
```

Creating a basic Windows PowerShell profile

The reason to learn about setting the Windows PowerShell script execution policy, as discussed in the preceding section, is that a Windows PowerShell profile is a script. In fact, it is a special script because it has a specific name and resides in a specific place. In this way, the Windows PowerShell profile is similar to the old-fashioned Autoexec.bat file: It has a special name, resides in a special location, and contains commands to customize the environment.

There are actually six different Windows PowerShell profiles, but for now you can create a Windows PowerShell profile for the current user and the current Windows PowerShell environment. The *$profile* automatic variable always points to the current user and current Windows PowerShell environment Windows PowerShell profile.

Determining if a Windows PowerShell profile exists

To see if a Windows PowerShell profile exists, use the *Test-Path* cmdlet. You can supply the *$profile* automatic variable to the *Test-Path* cmdlet when you check for the profile. If the profile exists, the *Test-Path* cmdlet returns *True*. If the profile does not exist, it returns *False*, as shown in the following example:

```
PS C:\> Test-Path $PROFILE
False
```

If a profile exists, you might want to back it up (for example, prior to creating a new one). To back up the profile, use the *Copy-Item* cmdlet and specify the *$profile* variable (containing the complete path to the profile) and the destination path. In the following example, the current user profile copies to a back-up file in the C:\fso directory.

```
Copy-Item $profile c:\fso\mycurrentbackupprofile.ps1
```

Creating a new Windows PowerShell profile

To create a new Windows PowerShell profile, use the *New-Item* cmdlet and specify the *$profile* automatic variable. Because the Windows PowerShell profile is a file (a script file), specify the *-ItemType* of file. The -Force parameter forces the cmdlet to create the file. The following example shows this technique:

```
PS C:\> New-Item $profile -ItemType file -Force


    Directory: C:\Users\ed.IAMMRED\Documents\WindowsPowerShell

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         11/22/2012  12:57 PM          0 Microsoft.PowerShell_profile.ps1
```

To edit the Windows PowerShell profile, open it in the Windows PowerShell ISE and add the *Start-Transcript* to the top of the Windows PowerShell profile. Click the Save icon and close the ISE. The following command opens the Windows PowerShell profile in the Windows PowerShell ISE:

```
PS C:\> ise $PROFILE
PS C:\>
```

Figure 2-7 shows the Windows PowerShell ISE and the *Start-Transcript* cmdlet on the first line.

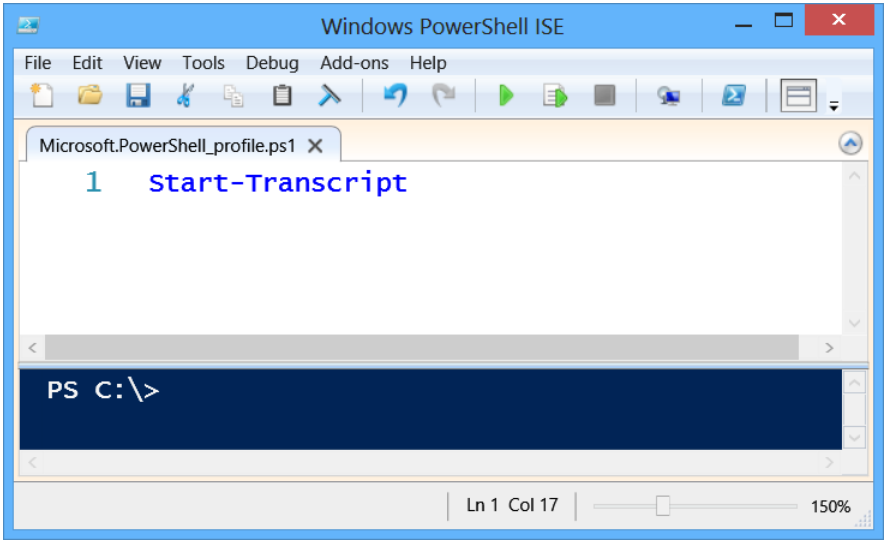


FIGURE 2-7 When you add the *Start-Transcript* cmdlet to the Windows PowerShell profile, a transcript of all commands and output from commands writes to the transcript log automatically.

Summary

This chapter focused on the basics of cmdlets. We reviewed how to search the Help topics, how to use the *Get-Command* cmdlet to find cmdlets, and how to use the *Get-Member* cmdlet to explore Windows PowerShell objects. We also reviewed how to set the Script Execution Policy and how to create a basic Windows PowerShell profile.

CHAPTER 3

Filtering, grouping, and sorting

- Introduction to the pipeline
- Sorting output from a cmdlet
- Grouping output after sorting
- Filtering output from one cmdlet
- Filtering output from one cmdlet before sorting

One of the tasks that Windows PowerShell excels at is providing insight into data. This usually involves sending data through the pipeline. In fact, the Windows PowerShell pipeline is a fundamental concept, and it is integral to sorting data, grouping data, and filtering out specific information from collections of other information. Using the Windows PowerShell pipeline is a skill you will use on a routine basis when working with Windows PowerShell.

Introduction to the pipeline

The Windows PowerShell pipeline takes the output from one command and sends it as input to another command. By using the pipeline, you can accomplish tasks such as finding all computers in one specific location and restarting them. You need one command to find all the computers in a specific location and another command to restart each of the computers. Passing the objects from one command to a new command makes Windows PowerShell easy to use inside the console because you do not have to stop and parse the output from the first command before taking action with a second command.

Windows PowerShell passes objects down the pipeline. This is one way that Windows PowerShell becomes very efficient: It takes an object (or group of objects) from the results of running one command and it passes those objects to the input of another command. By using the Windows PowerShell pipeline, it is not necessary to store the results of one

command into a variable and then call a method on that object to perform an action. For example, the following command disables all network adapters on my Windows 8 computer:

```
Get-NetAdapter | Disable-NetAdapter
```

> ***NOTE*** **Windows PowerShell honors the security policy. Therefore, to disable a network adapter, you must run Windows PowerShell with administrator rights. For more information about starting Windows PowerShell with administrator rights, see Chapter 1.**

In addition to disabling all network adapters, you can enable them as well. To do this, use the *Get-NetAdapter* cmdlet and pipeline the results to the *Enable-Netadapter* cmdlet, as shown in the following example:

```
Get-NetAdapter | Enable-NetAdapter
```

If you want to start all the virtual machines on Windows 8 or Windows Server 2012, use the *Get-VM* cmdlet and pipeline the resulting virtual machine objects to the *Start-VM* cmdlet, as shown in the following example:

```
Get-VM | Start-VM
```

To shut down all the virtual machines, use the *Get-VM* cmdlet and pipeline the resulting virtual machine objects to the *Stop-VM* cmdlet, as shown in the following example:

```
Get-VM | Stop-VM
```

In each of the preceding commands, an object (or group of objects) resulting from one command pipelines to another cmdlet for further action.

Sorting output from a cmdlet

The *Get-Process* cmdlet generates a nice table view of process information to the Windows PowerShell console. The default view appears in ascending alphabetical process name order. This view is useful for helping to find specific process information, but it hides important details, such as which process uses the least, or the most virtual, memory. To sort the output from the process table, pipeline the results from the *Get-Process* cmdlet to the *Sort-Object* cmdlet and supply the property upon which to sort to the -Property parameter. The default sort order is ascending (that is, smallest numbers appear at the top of the list). The following command sorts the process output by the amount of virtual memory used by each process:

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

The processes consuming the least amount of virtual memory appear at the top of the list.

If you are interested in which processes consume the most virtual memory, you might want to reverse the default sort order. To do this, use the -Descending switch parameter, as shown in the following example:

```
Get-Process | Sort-Object -Property VM -Descending
```

Figure 3-1 shows the command to produce the virtual memory sorted list of processes and the associated output from the command.

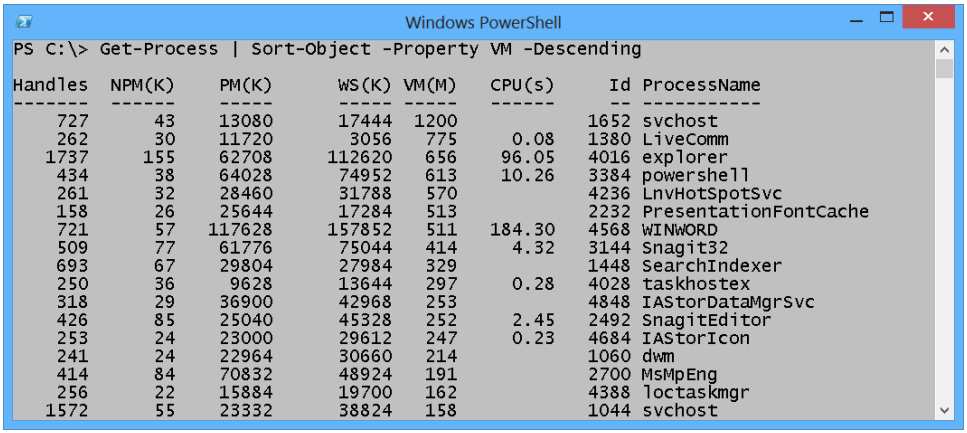


FIGURE 3-1 Use the *Sort-Object* cmdlet to organize object output into readable output.

It is possible to shorten the length of Windows PowerShell commands that use the *Sort-Object* cmdlet. The command *Sort* is an alias for the *Sort-Object* cmdlet. A cmdlet alias is a shortened form of the cmdlet name that Windows PowerShell recognizes as a substitute for the complete cmdlet name. Some aliases are easily recognizable, such as sort for *Sort-Object* or select for *Select-Object*. Other aliases must be learned, such as *?* (the question mark symbol) for the *Where-Object* cmdlet. Most Windows users expect *?* to be an alias for the *Get-Help* cmdlet.

In addition to using an alias for the *Sort-Object* cmdlet name, the -Property parameter is the default parameter the cmdlet uses, so it can be omitted from the command. The following command uses the shortened syntax to produce a list of services by status:

```
Get-Service | sort status
```

It is possible to sort on more than one property. You need to be careful doing this because at times it is not possible to sort additional properties. With the services, a multiple sort makes sense because there are two broad categories of status: Running and Stopped. It therefore makes sense to attempt to organize the output further to facilitate finding particular stopped or running services. One way to facilitate finding services is to sort alphabetically the displayname property of each service. The following example sorts the service objects obtained through the *Get-Service* cmdlet by the status, and then by the *displayname* from within the status:

```
Get-Service | sort status, displayname -Descending
```

The output appears in descending order instead of the default ascending sorted list.

Figure 3-2 shows the command to sort services by status and *displayname* as well as the output from the command.

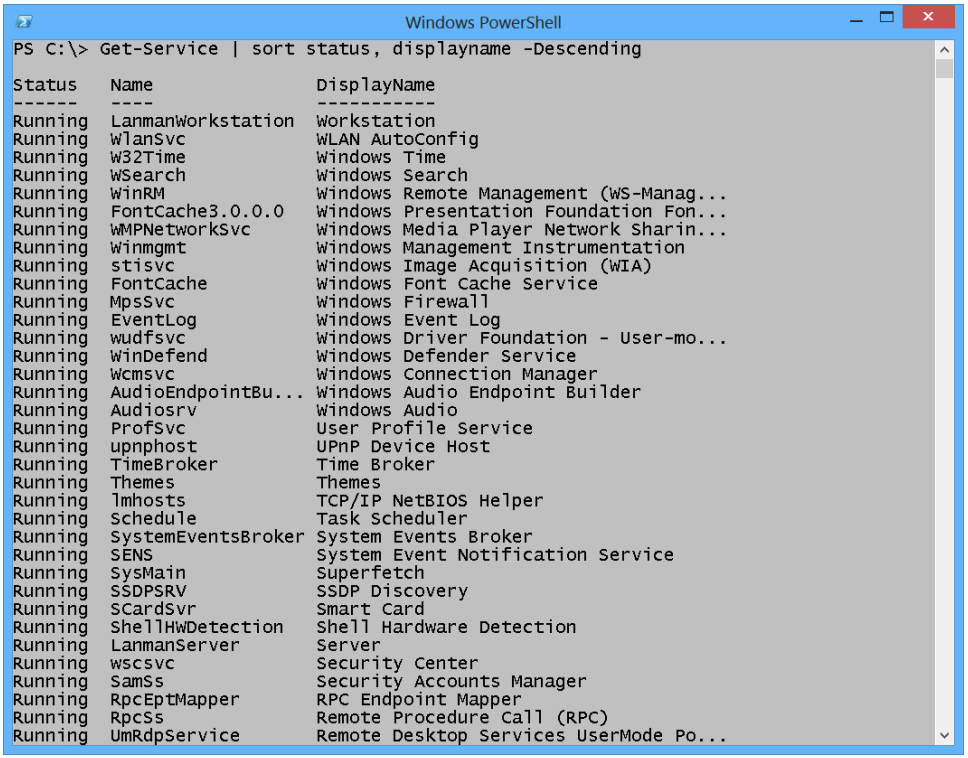


FIGURE 3-2 Using the *Sort-Object* cmdlet to organize the output from the *Get-Service* cmdlet.

Grouping output after sorting

After you have sorted the objects coming through the pipeline, you can then group them. It is important to sort the objects prior to grouping to help to ensure the best performance and the most accurate results. To group the count of running or stopped services, use the *Get-Service* cmdlet to retrieve the service objects. Pipeline the resulting service objects to the *Sort-Object* cmdlet and sort on the status property. Finally, pipeline the sorted service objects to the *Group-Object* cmdlet and specify that you want to group on the status property. The following example shows the resulting command and associated output:

```
PS C:\> Get-Service | Sort-Object status  | Group-Object -Property status

Count Name                      Group
----- ----                      -----
   99 Stopped                   {PNRPsvc, p2pimsvc, ose, TrustedInstaller...}
   83 Running                   {vmms, wudfsvc, Wcmsvc, stisvc...}
```

NOTE **When you use the *Group-Object* cmdlet, it is vital that you specifically select the property on which to group. The property on which to group should be the property on which you sorted. Without specifying an object on which to group, the command appears to work but the results are inconsistent.**

You can shorten the length of the *Group-Object* cmdlet by using the *Group* cmdlet alias instead of typing out *Group-object*. In addition, the -Property parameter is the default parameter and can be omitted. The following example shows the shortened version of the command to display running and stopped service counts:

```
PS C:\> Get-Service | sort status | group status

Count Name                      Group
----- ----                      -----
   95 Stopped                   {PNRPsvc, p2psvc, p2pimsvc, ose...}
   87 Running                   {wudfsvc, SysMain, Wcmsvc, wuauserv...}
```

Grouping information without element data

By default, the *Group-Object* cmdlet displays three properties: *Count*, *Name*, and *Group*. The *Group* property contains data associated with the values appearing under the *Name* property. Often the data in the Group column is useful because it provides examples of the kind of values available for the grouping. At times, however, this output is distracting. The preceding examples of using the *Group-Object* cmdlet contain a column that provides a small clue as to the items grouped. By default, Windows PowerShell displays four items in the Group column, which generally is enough to provide an indication as to how the command ran. However, it is also a bit of a distraction having the column with the braces as well as the ellipses cluttering up the screen. When you work with the event log, for example, the group data displays the data type of the entry. This information is similar for all event log entries, and therefore is spurious. The following example shows the five most recent error events from the application log:

```
12:47 C:\> Get-EventLog -Log application -EntryType error -new 5| sort message |
group message

Count Name                      Group
----- ----                      -----
    1 (GetHomepage()): Faile... {System.Diagnostics.EventLogEntry}
    1 (GetHomepages()): Fail... {System.Diagnostics.EventLogEntry}
    1 (PerformTasks()): Fail... {System.Diagnostics.EventLogEntry}
    2 App DefaultBrowser_NOP... {System.Diagnostics.EventLogEntry, System.Diagnos...
```

The five error messages are sorted by the message value and displayed to the Windows PowerShell console. Notice that the Group column occupies value display space, and the data that provides the most important information, the Name column, is truncated to the point that the information is barely usable.

If you know your command will return the information you seek, and you are interested only in the group counts, you can use the *Group-Object* cmdlet to return only the grouped property and the count of the items in that group. The key is to use the -NoElement switched parameter. The command that follows groups the processes by name, then sorts the count of the processes in descending order:

```
Get-Process | sort name | group name -NoElement | sort count -Descending
```

Figure 3-3 shows the command to get running processes, sort by name, group the names, and sort the count of the running processes by name. Figure 3-3 also shows the output from the command.

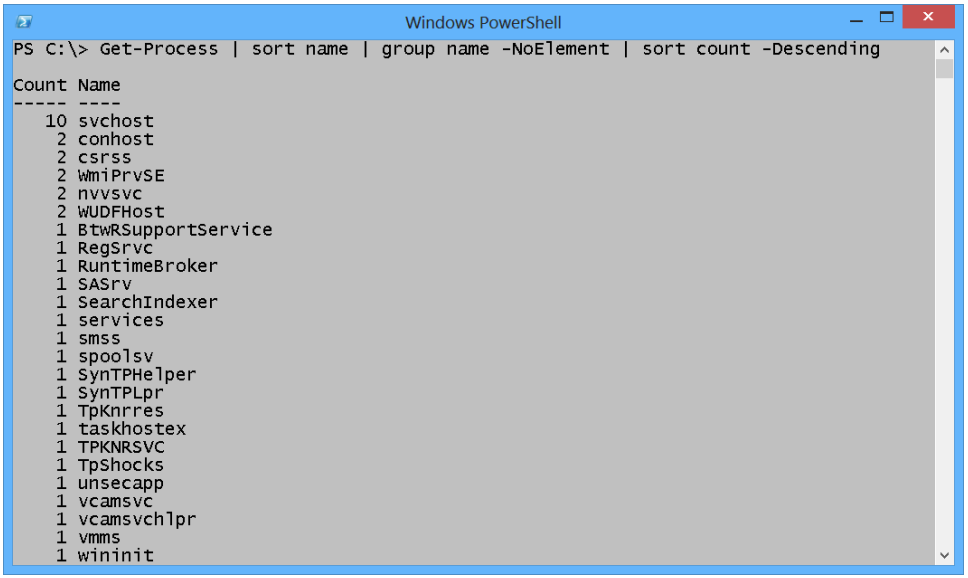


FIGURE 3-3 By using the -NoElement switched parameter, the grouping data does not display to the Windows PowerShell console.

Filtering output from one cmdlet

Grouping and sorting data is useful in that it permits a general overview of the queried data. However, it is data filtering that permits you to dive into the data and surface relevant patterns. For example, you might spend a long time staring at the output from the *Get-Process* cmdlet before you find there is a process on your system that is using 1,000 MB of virtual memory. But with a simple *Where-Object* filter, the information becomes blatantly obvious. The following example shows the command to retrieve process information and to filter out all processes that are using more than 1,000 MB of virtual memory:

```
PS C:\> Get-Process | Where-Object vm -gt 1000MB

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    762      46    12824      17140  1203            1656 svchost
```

Filtering by date

Many times, from a troubleshooting perspective, you need to look at what happened after a specific date. Some cmdlets permit direct filtering by date while others do not. If a cmdlet does not have a parameter that permits filtering by date, all that is required is to pipeline the returned objects to the *Where-Object* cmdlet.

The *Get-WindowsDriver* function from the Deployment Image Servicing and Management (DISM) module in Windows 8 retrieves a listing of all the installed drivers on a computer. While there are several parameters available to filter drivers on the left side of the pipeline character, there is no facility to filter by date. To find drivers installed after a specific date, pipeline the results of *Get-WindowsDriver* to the *Where-Object* and specify that you want only drivers installed after a specific date. The following example shows this command:

```
PS C:\> Get-WindowsDriver -Online | where date -gt 10/8/2012

Published Name     : oem26.inf
Original File Name : C:\Windows\System32\DriverStore\FileRepository\ibmpmdrv.inf_amd
                     64_728348017c675c91\ibmpmdrv.inf
InBox              : False
Class Name         : System
Boot Critical      : True
Provider Name      : Lenovo
Date               : 10/9/2012 12:00:00 AM
Version            : 1.66.0.17
```

> ***NOTE*** **You need elevated rights to use the Windows 8 *Get-WindowsDriver* function.**

The *Sort-Object* cmdlet works great with dates. Sorting dates makes identifying similar events easy. For example, the following command returns all hotfixes installed after December 1, 2012 on a local computer:

```
Get-HotFix | Where installedon -gt 12/1/12
```

The only problem with the command to list hotfixes installed after December 1, 2012 is that a lot of hotfixes were released in December, 2012. This makes finding a particular hotfix a problem. However, if you pipeline the results to the *Sort-Object* cmdlet, you can identify specific hotfixes easily. In fact, the default-ascending sort is perfect for date-related sorting jobs. The following example shows the command:

```
Get-HotFix | Where installedon -gt 12/1/12 | sort installedon
```

Figure 3-4 shows the command to find and sort all hotfixes installed after December 1, 2012 and the associated output from the command.

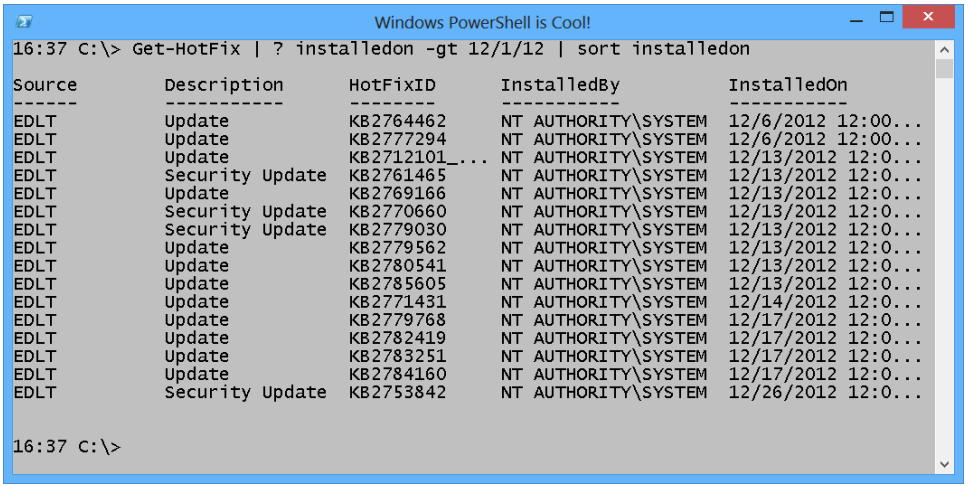


FIGURE 3-4 Sorting dates after filtering by date is an easy way to identify patterns in data.

The object that returns dates is called the *DateTime* object. The *DateTime* object is made up of a number of methods and properties. The properties of the *DateTime* object are what are used when you perform a filter. To see the properties of the *DateTime* object, pipeline a date to the *Get-Member* cmdlet. The following example shows the command to return the properties of a *DateTime* object and the output from the command:

```
16:59 C:\> Get-Date | Get-Member -MemberType Property

   TypeName: System.DateTime

Name        MemberType Definition
----        ---------- ----------
Date        Property   datetime Date {get;}
Day         Property   int Day {get;}
DayOfWeek   Property   System.DayOfWeek DayOfWeek {get;}
DayOfYear   Property   int DayOfYear {get;}
Hour        Property   int Hour {get;}
Kind        Property   System.DateTimeKind Kind {get;}
Millisecond Property   int Millisecond {get;}
Minute      Property   int Minute {get;}
Month       Property   int Month {get;}
Second      Property   int Second {get;}
Ticks       Property   long Ticks {get;}
TimeOfDay   Property   timespan TimeOfDay {get;}
Year        Property   int Year {get;}
```

The time that a process starts is reported as a *DateTime* object. The *Get-Member* cmdlet reveals this information through the following command:

```
17:03 C:\> Get-Process | Get-Member -Name starttime

   TypeName: System.Diagnostics.Process

Name      MemberType Definition
----      ---------- ----------
StartTime Property   datetime StartTime {get;}
```

Because the StartTime property contains an instance of the *DateTime* object, any property from that object can be used to create a filter. To filter the time by hour, minute, second, or any other property from the *DateTime* objects requires using a more complicated form of the *Where-Object* cmdlet. This form begins and ends with a *ScriptBlock*. A pair of braces ({}) marks the *ScriptBlock*. Once inside the *ScriptBlock*, the *$_* automatic variable is used to gain access to each item as it crosses the pipeline. From there, it works like any other filter. Choose the property from the object, and use the greater than (>) operator to specify which minute to filter on, as shown in the following example:

```
17:06 C:\> Get-Process | Where { $_.starttime.minute -gt 55}  | select name, starttime

Name                                       StartTime
----                                       ---------
IAStorIcon                                 1/5/2013 2:56:32 PM
WINWORD                                    1/5/2013 2:58:25 PM
```

Filtering to the left

From a performance perspective, it's best to perform most actions to the data on the left side of the pipeline character. This is because when a cmdlet runs, it might return a lot of data. Even when you work locally, returning large amounts of data could involve significant memory, CPU time, and disk I/O. If you return data from across the network, then an inefficient query also causes network performance issues. In general, when you work locally (or when you work remotely with fast network connections) there is a point of diminishing returns. For example, if you spend 10 minutes perfecting a query that takes only a few seconds to run (when not optimized), you have probably wasted 9.9 minutes that could have been more effectively utilized. If, on the other hand, the non-optimized query takes 10 minutes to run, and you are going to run it on 100 remote servers, some of which are severely bandwidth constrained, then you would be justified in spending several days to optimize the query.

So how do you filter to the left of the pipeline character? You use the filtering capabilities of the cmdlet itself. For example, the following command returns all the event log entries from the application log:

```
Get-EventLog -LogName application
```

On my computer, there are 10,575 event log entries in the application log. If you are interested in looking at only error logs, you can use the *Where-Object* cmdlet, as shown in the following example:

```
Get-EventLog -LogName application | where entrytype -eq 'error'
```

On my computer, the previous command takes 1.8 seconds to complete. If I use the *-EntryType* filter from the *Get-EventLog* cmdlet, I arrive at the following query:

```
Get-EventLog -LogName application -EntryType error
```

By filtering to the left of the pipeline (in this case, not piping to the *Where-Object*) the command runs in .8 seconds, which is less than half the amount of time. Now, to be truthful, when I work on my local computer I am not too concerned about reducing a command from 1.8 seconds to .8 seconds. However, you will also note that by using the -EntryType parameter from the *Get-EventLog* cmdlet, the command is shorter and easier to read. In addition, it is less error prone because tab expansion is used to complete the command.

Filtering output from one cmdlet before sorting

When you know you are going to sort your output, it is better to reduce the amount of data to sort before performing the sort operation. Earlier, I discussed the dictum "filter to the left." The corollary is to "sort on the right." The sort operation is more efficient when it has fewer items to sort. The Windows 8 cmdlet *Get-AppxPackage* returns a lot of data. In fact, it returns so much data, it can be cumbersome to use. Figure 3-5 shows the unfiltered use of the *Get-AppxPackage* cmdlet.

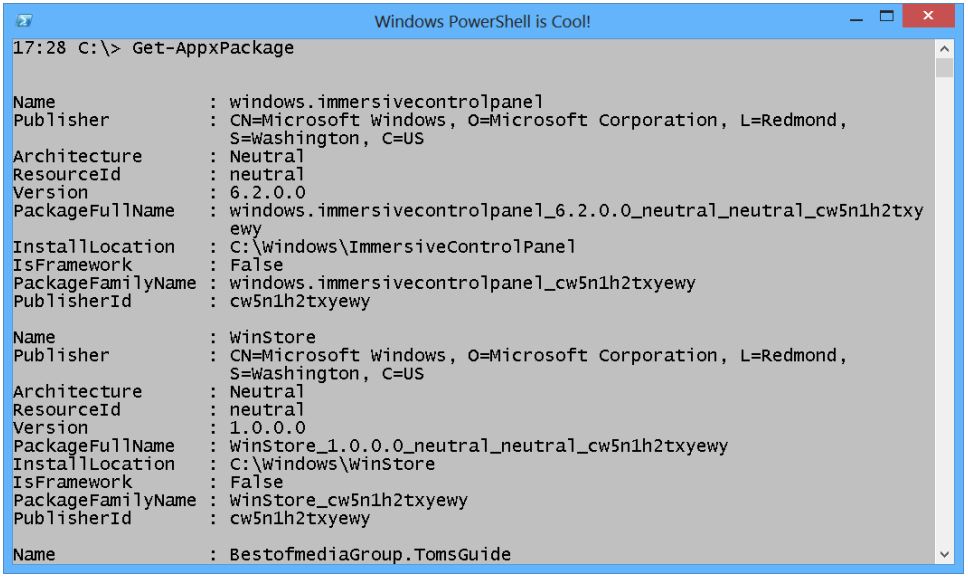


FIGURE 3-5 Because of the amount of data returned by the *Get-AppxPackage* cmdlet, it is difficult to find specific information about packages from a specific publisher.

To find specific information about packages from a specific publisher, it is better to choose only the properties that are interesting to you and then sort on the most important property. In the following example, only the *Name*, *Version,* and *Publisher* properties are chosen from the *AppxPackage* objects returned by the *Get-AppxPackage* cmdlet. Once the properties are selected, the *Where-Object* cmdlet is used to filter resulting packages created by Microsoft. Finally, the names of the packages are sorted in an ascending list. The following example shows the command to return a filtered list of packages:

```
Get-AppxPackage | Select Name, Version, Publisher | Where Publisher -Match Microsoft |
Sort Name
```

Figure 3-6 shows the output from the filtered list of packages, along with the command creating the list.

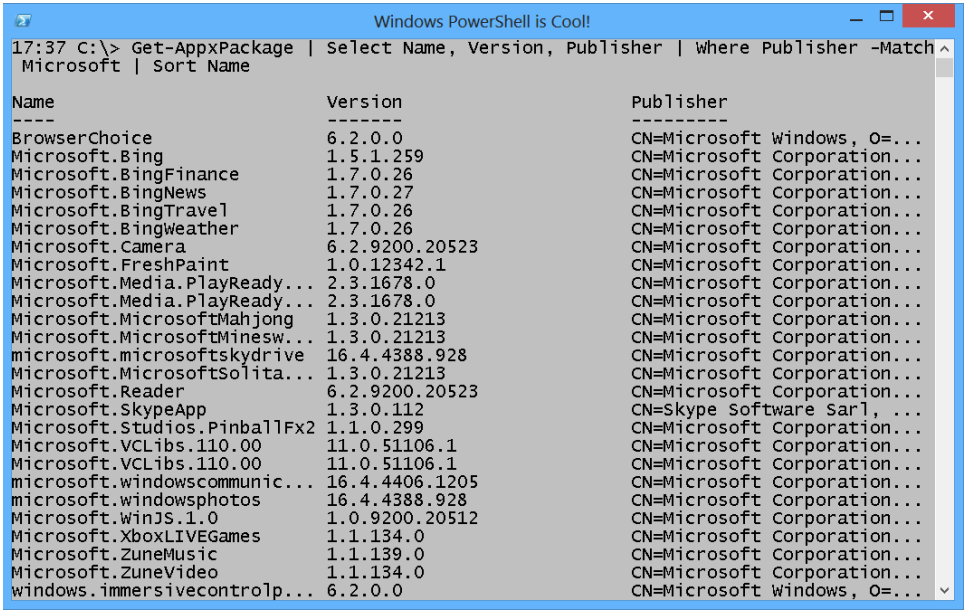


FIGURE 3-6 By filtering the output from a command prior to sorting the output, a clean, easily read output displays.

Summary

This chapter began with an introduction to the pipeline. Next, it covered sorting output from a cmdlet, grouping output after sorting, and filtering output from one cmdlet. The chapter concluded with filtering output from one cmdlet before sorting.

CHAPTER 4

Formatting output

- Creating a table
- Creating a list
- Creating a wide display
- Creating an output grid

When you work with Windows PowerShell, it is common to want to format the output to the Windows PowerShell console. This is not always a requirement, however, due to the fact that many Windows PowerShell cmdlets include their own formatted output. For example, the *Get-Process* cmdlet produces a nice table output that readily meets the needs for 90 percent of those who require process information. Occasionally, however, it becomes necessary to customize the output. In this chapter, you learn about creating tables, lists, wide lists, and even how to use a selectable grid view.

> ***NOTE*** **If you use any of the formatting tools mentioned in this chapter, they must appear at the end of the Windows PowerShell pipeline. Once you send output to a formatter, you can no longer manipulate the data. If you attempt to sort, filter, or group data after the formatting cmdlets, an error arises.**

Creating a table

When you have between two and five properties you are interested in viewing in columns of data, the *Format-Table* cmdlet is the tool to use to organize your data. The typical use of *Format-Table* is to permit delving into specific information in a customizable way. For example, the *Get-Process* cmdlet returns a table with eight columns containing essential process information. Figure 4-1 shows the *Get-Process* command and the resulting output.

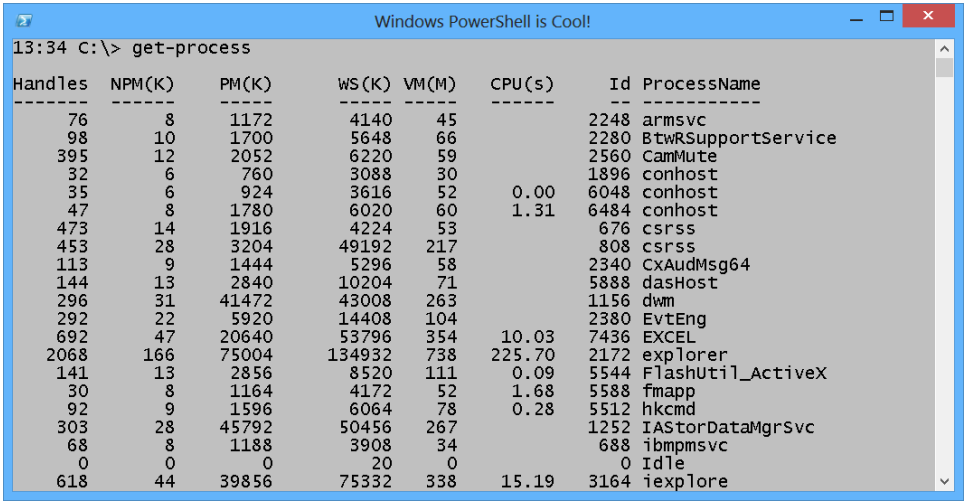


FIGURE 4-1 The *Get-Process* cmdlet returns an eight-column table by default.

Choosing specific properties in a specific order

If the eight columns of default process information meet your needs, there is no need to think about using a formatting cmdlet. However, the *Process* object returned by the *Get-Process* cmdlet actually contains 51 properties and 7 script properties. As a result, there is much more information available than just the eight default properties. To dive into this information requires using one of the formatting cmdlets. From the perspective of the *Get-Process* cmdlet, there are six alias properties. Alias properties are great because they can shorten the amount of typing required. The *Get-Process* alias properties appear in the following output:

```
13:40 C:\> get-process | get-member -MemberType alias*

   TypeName: System.Diagnostics.Process

Name    MemberType    Definition
----    ----------    ----------
Handles AliasProperty Handles = Handlecount
Name    AliasProperty Name = ProcessName
NPM     AliasProperty NPM = NonpagedSystemMemorySize
PM      AliasProperty PM = PagedMemorySize
VM      AliasProperty VM = VirtualMemorySize
WS      AliasProperty WS = WorkingSet
```

To use the *Format-List* cmdlet, you pipeline the results from one cmdlet to the *Format-List* cmdlet and select the property names you want to display.

> ***NOTE*** **The order in which the properties appear is the order in which they display in the table.**

The following command displays process information from every process on the local system. The specified properties use the alias properties created for the *Get-Process* cmdlet. The output is in the order of Name, Handles, Virtual Memory Size, and the Working Set.

```
Get-Process | Format-Table -Property name, handles, vm, ws
```

Figure 4-2 shows the command to produce the formatted list of process information as well as the output associated with the command.

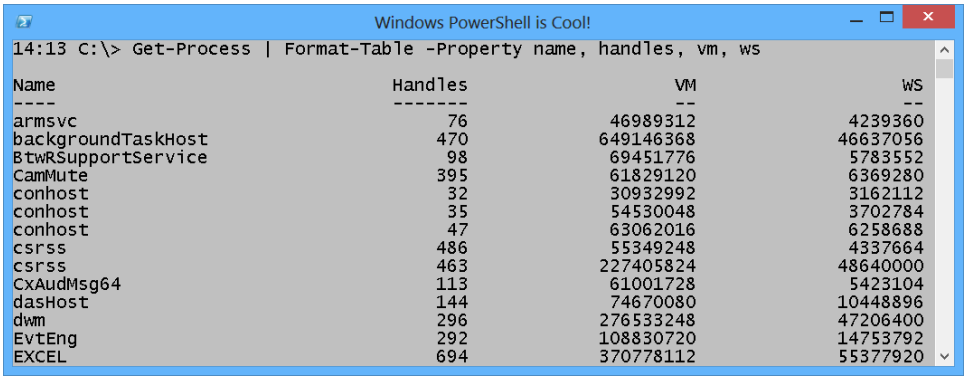


FIGURE 4-2 Using the *Format-Table* cmdlet, you can specify the order of selected properties from a command.

NOTE **The *Get-Process* cmdlet has an alias of *GPS*, and the *Format-Table* cmdlet has an alias of *FT*. Therefore, the command to return a table of process information can be shortened to the following:**

```
GPS | FT name, handles, vm, ws
```

Controlling the way the table displays

When you pipeline information to the *Format-Table* cmdlet, it has to determine how much space to reserve for each column. What happens is that the *Format-Table* cmdlet takes a quick look at the data as it crosses the pipeline and makes a guess that is based upon the number of columns, the width of the Windows PowerShell console, and the length of each value to display in each column. For commands that send vast quantities of data across the pipeline, the actual width of the columns may shift as more and more data becomes available. Because this quick look at the data seldom produces a completely optimized display, you might want to have the *Format-Table* cmdlet wait until all the data crosses the pipeline, and then determine the maximum amount of space to permit each column. While this results in a table that optimizes the amount of available space, it does add to the amount of time the command takes to run. Often, the good enough display is just that: It is good enough, and speedy return of the data is more important than a perfectly formatted and optimized output. The

following command returns the service names, the status of each service, and the display name of services defined on the local system:

```
Get-Service | Format-Table -property name, status, displayname
```

Figure 4-3 shows the output after the command runs.

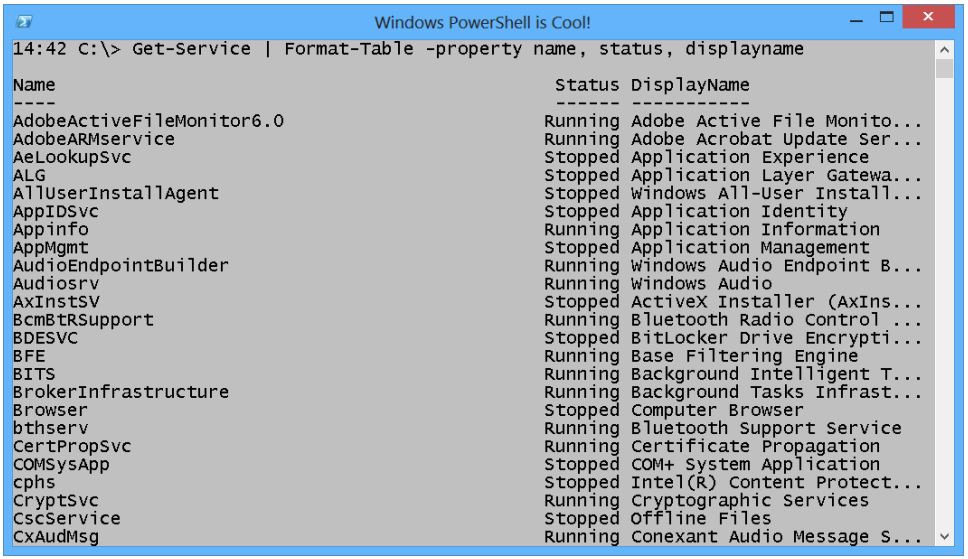


FIGURE 4-3 Using *Format-Table* defaults results in a quick but non-optimized output.

To fix the output appearing in Figure 4-3, you need to add the -AutoSize parameter to the end of the *Format-Table* command. The -Autosize parameter causes *Format-Table* to wait until all data is available, and then the space between columns reduces to fit the actual size of the data contained in the columns. The following example shows the revised command:

```
Get-Service | Format-Table -property name, status, displayname -AutoSize
```

Figure 4-4 shows the command to display service names, status, and display names of all defined services.

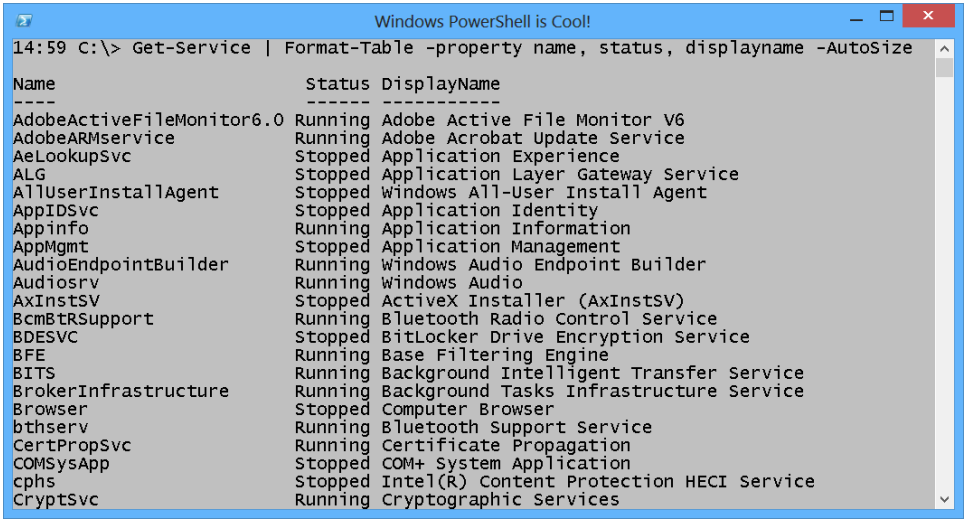


FIGURE 4-4 Using the -AutoSize parameter reduces wasted space between columns and produces a more readable output.

The output in Figure 4-4 is much better than the previous output because of the optimized space between columns that results in displaying maximum data. However, there are still a couple of service display names that are too long to display in the limited amount of console output. When this happens, the output truncates. The following two lines of code illustrate the truncated service display names:

```
LENOVO.TPKNRSVC          Running Lenovo AVFramework Microphone Volume Controller...
LENOVO.TVTVCAM           Running Lenovo AVFramework Control Center and ThinkVant...
```

To display the overly long display names, use the -Wrap parameter in addition to using the -AutoSize parameter. The following example shows the revised code:

```
Get-Service | Format-Table -property name, status, displayname -AutoSize -wrap
```

Once the revised command runs, the two previously truncated Lenovo service display names wrap to the subsequent lines:

```
LENOVO.TPKNRSVC          Running Lenovo AVFramework Microphone Volume Controller
                                 and Dolby Interface
LENOVO.TVTVCAM           Running Lenovo AVFramework Control Center and
                                 ThinkVantage Virtual Camera Controller
```

Creating a list

When you want to see all the properties and associated values returned by a particular command, using *Format-List* is the easy way to display the information. By using a wildcard to select all properties from an object, the properties and associated values of the properties appear on individual lines in the Windows PowerShell console. Figure 4-5 shows the output from the *Get-Process* cmdlet when pipelined to the *Format-List* cmdlet.

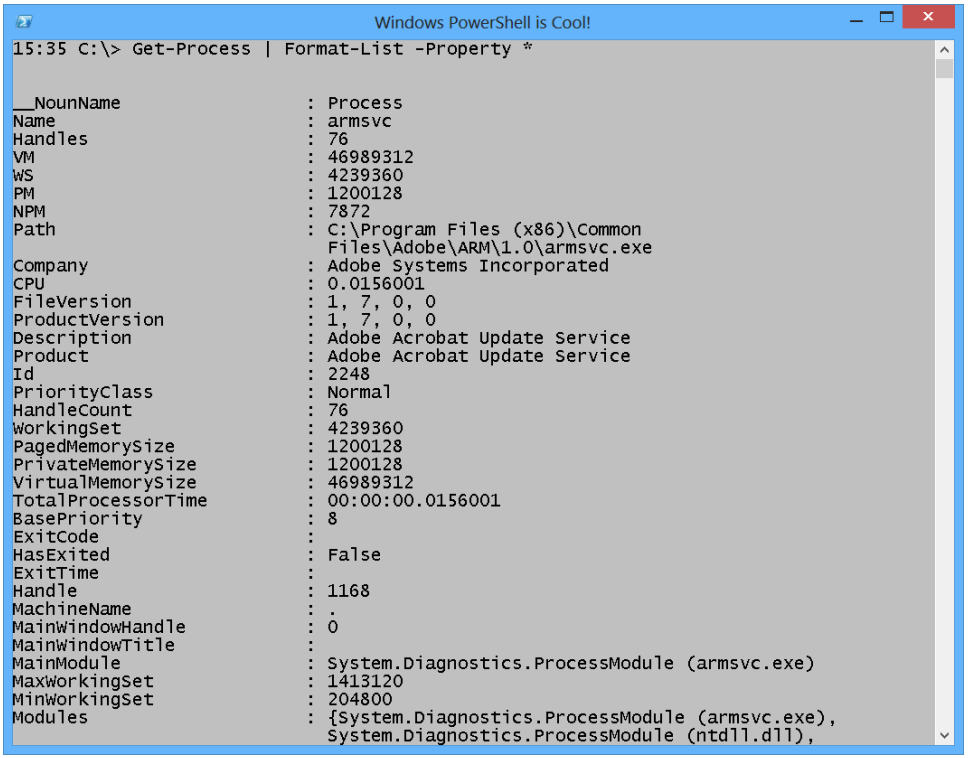


FIGURE 4-5 Use a wildcard character to select all properties to display when pipelining results to the *Format-List* cmdlet.

> ***NOTE*** **When using the *Get-Process* cmdlet, a non-elevated user does not have access to all properties and their associated values. For example, the path parameter will not display information to a non-elevated user. The output in Figure 4-5 was produced by right-clicking on the Windows PowerShell button and selecting Run as Administrator from the Task menu.**

Choosing properties by name

You are not limited to only displaying all properties from a command. You can choose properties by name, and the resulting output will still display one property/value pair per line. The following example chooses the first five entries from the application log on the local computer. It then selects the source, entry type, and event ID from each event log entry:

```
Get-EventLog -LogName application -Newest 5 | Format-List -property source, entrytype,
eventID
```

Figure 4-6 shows the command to display the source, entry type, and event ID from the first five entries in the application log on the local computer, along with the associated output.

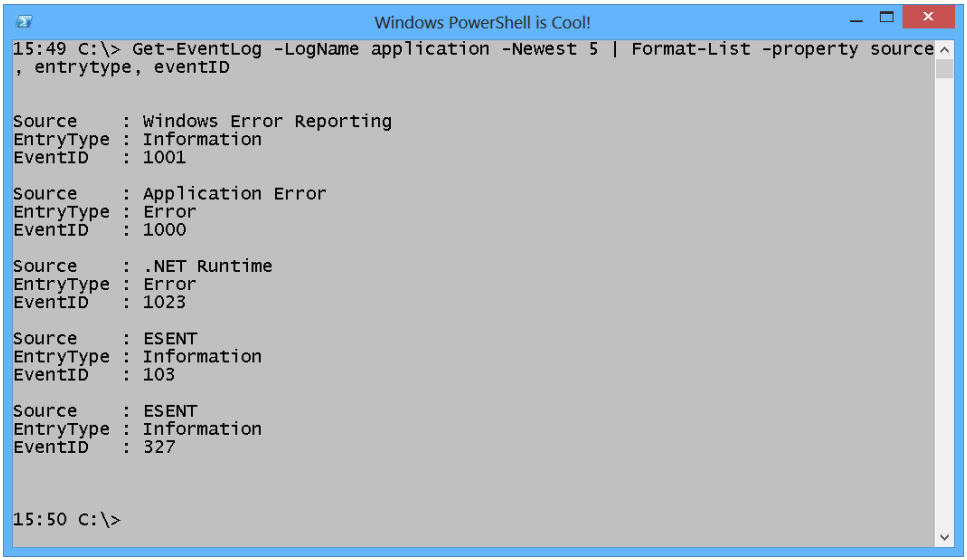


FIGURE 4-6 *Format-List* cmdlet produces a nice, tight display for information that would be too long to fit easily in a table.

Choosing properties by wildcard

The command shown in the preceding Figure 4-6 is rather long and reduces some of the efficiency obtained by working directly in the Windows PowerShell console. To shorten the command, use the fl alias for the *Format-List* cmdlet. Also, it is not necessary to specify the -Property parameter because it is the default parameter for the *Format-List* cmdlet. But the real power comes from using wildcards to select the properties to display. By supplying just a few characters, the desired properties display. The following example shows the revised command. (The default parameter for the *Get-EventLog* cmdlet is -LogName, so it can be left out.) Instead of typing the complete -Newest parameter, only the first letter is required:

```
Get-EventLog application -N 5 | Fl s*,e*
```

When the revised command runs, it adds an additional property, the Site property. The following example shows the command and associated output from the command:

```
16:08 C:\> Get-EventLog application -N 5 | Fl s*,e*


Source    : ESENT
Site      :
EventID   : 910
EntryType : Warning

Source    : ESENT
Site      :
EventID   : 910
EntryType : Warning

Source    : ESENT
Site      :
EventID   : 910
EntryType : Warning

Source    : ESENT
Site      :
EventID   : 910
EntryType : Warning

Source    : ESENT
Site      :
EventID   : 910
EntryType : Warning
```

To keep the additional Site property from displaying, add one letter to the *Format-List* command. The following example shows this revision with the accompanying desired output:

```
16:09 C:\> Get-EventLog application -N 5 | Fl so*,e*


Source    : ESENT
EventID   : 910
EntryType : Warning

Source    : ESENT
EventID   : 910
EntryType : Warning

Source    : ESENT
EventID   : 910
EntryType : Warning

Source    : ESENT
EventID   : 910
EntryType : Warning

Source    : ESENT
EventID   : 910
EntryType : Warning
```

Creating a wide display

If you are interested in a single property, use the *Format-Wide* cmdlet. If you are interested only in a listing of the names of processes, *Format-Wide* is the ideal choice. To display the name of each process, use the *Get-Process* cmdlet to obtain the process objects. Pipeline the results to the *Format-Wide* cmdlet and specify Name for the -Property parameter:

```
Get-Process | Format-Wide -Property name
```

The resulting output from the *Format-Wide* cmdlet appears to be a two-column table, but because it contains only a single property, it actually is a single property displaying in two columns. Figure 4-7 shows the output.

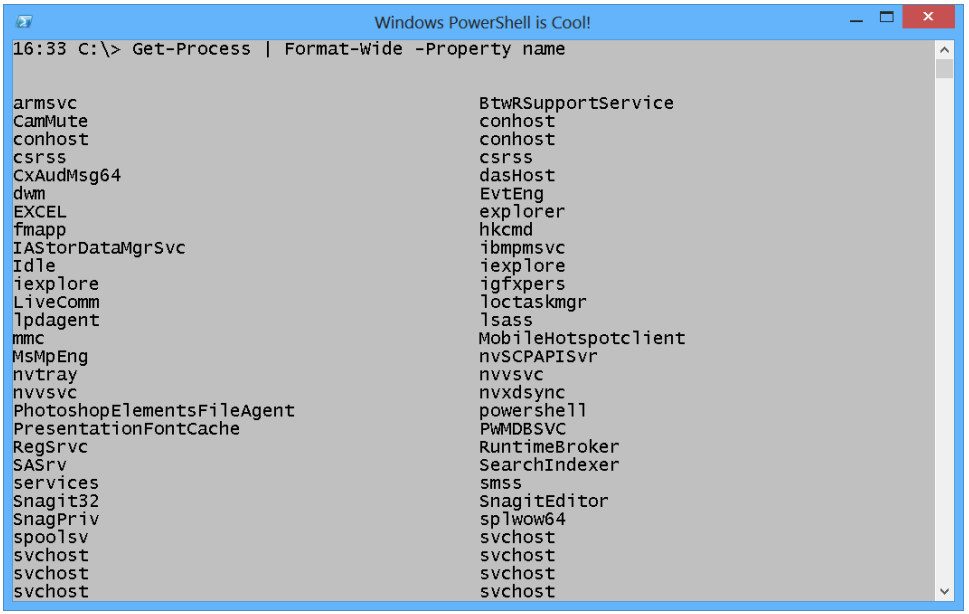


FIGURE 4-7 A two-column display created by the *Format-Wide* cmdlet.

Using the -AutoSize parameter to configure the output

By default, the *Format-Wide* cmdlet works in a similar fashion to the *Format-Table* cmdlet; they both sort of guess at the proper spacing. This means that you will generally get a rather loose two-column display. To tighten up the display, use the -AutoSize parameter. Just like with the *Format-Table* cmdlet, using -AutoSize increases the time required to produce output. This is because all the data must be obtained, analyzed, and the number of columns chosen to avoid truncating output.

The following command obtains information from all processes on the local computer. The resulting process objects are piped to the *Format-Wide* cmdlet where the *Name* property is selected. The -AutoSize switched parameter is used to tighten up the display:

```
Get-Process | Format-Wide -Property name -AutoSize
```

Figure 4-8 shows the *Get-Process* command and the output associated with the *Format-Wide* cmdlet.

FIGURE 4-8 Using the -AutoSize parameter with the *Format-Wide* cmdlet tightens up the output.

Customizing the *Format-Wide* output

It might be that the output produced by using the -AutoSize parameter still takes up too much space. For example, in the preceding Figure 4-8, some of the process names are rather lengthy. The longest process name is 26 characters in length. The next longest one is 21 characters long. In addition to the few abnormally long process names, if the names were truncated just a bit, enough of the name would still be visible so you could make an intelligent decision as to the process. For example, the *Format-Wide* cmdlet displays the names of processes on the local computer, with four columns. However, only two process names truncate, as shown in Figure 4-9.

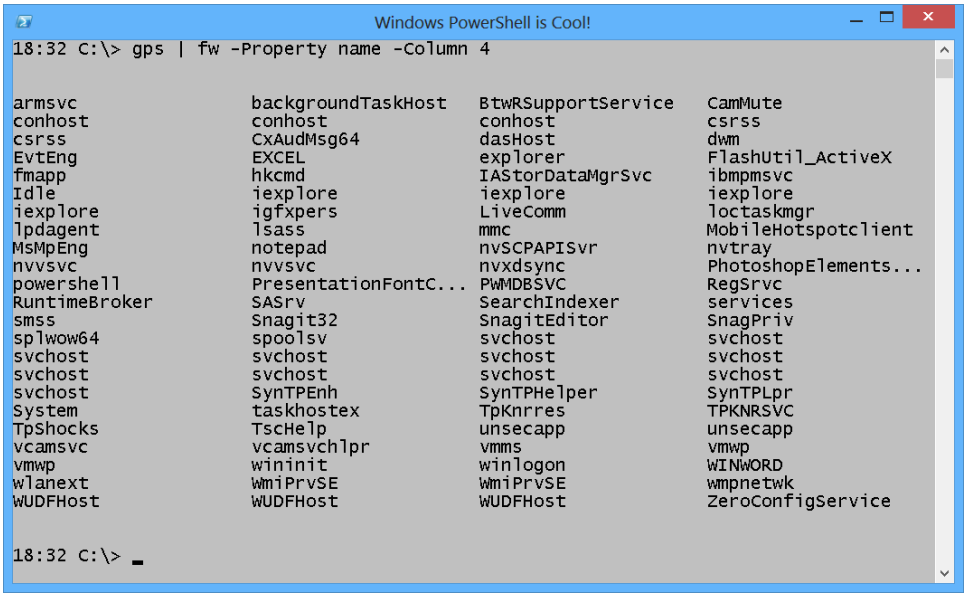


FIGURE 4-9 By specifying an exact number of columns with *Format-Wide*, you can control the amount of truncation that occurs.

By changing the -Column parameter to 5, the number of truncated process names increases to eight, but the number of rows decreases to 19 from the previous 23. Therefore, it is up to you how compact the output can be. Keep in mind that the -Autosize parameter and the -Column parameter are mutually exclusive. You cannot tell Windows PowerShell to automatically size the *Format-Wide* output and then try to tell it how many columns to use.

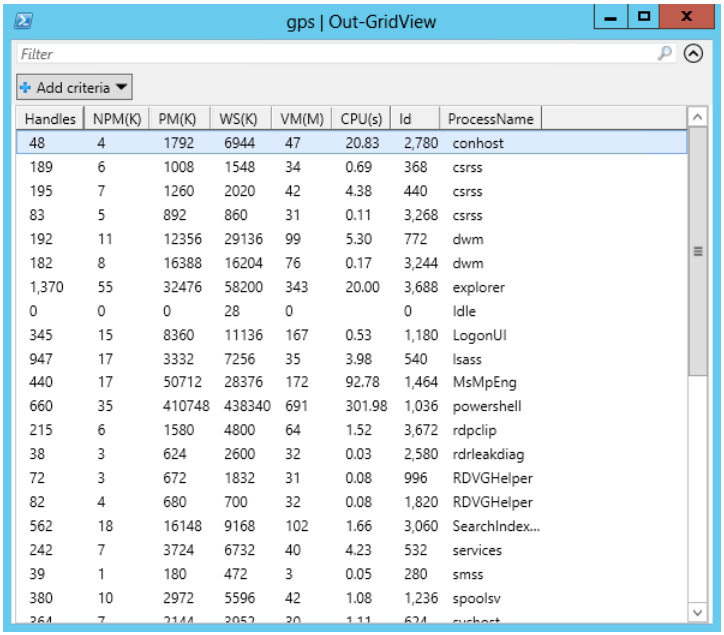
Creating an output grid

The *Out-GridView* cmdlet is different from the other formatting cmdlets explored thus far in this chapter. The *Out-GridView* cmdlet is an interactive cmdlet; that is, it does not format output for display on the Windows PowerShell console or for sending to a printer. Instead, *Out-GridView* provides a control permitting exploration of the pipelined data. For example, the following command pipelines the results of the *Get-Process* cmdlet to the *Out-GridView* cmdlet (*gps* is an alias for the *Get-Process* cmdlet):

```
gps | Out-GridView
```

Sorting output by using the column buttons

When the command completes, a grid appears containing process information arranged in columns and in rows. The new window displaying the process information in a grid appears in Figure 4-10. One useful feature of the *Out-GridView* cmdlet is that the returned control contains the command producing the control in the title bar. Figure 4-10 lists the command `gps | Out-GridView` in the title bar (the command run to produce the grid control).



Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
48	4	1792	6944	47	20.83	2,780	conhost
189	6	1008	1548	34	0.69	368	csrss
195	7	1260	2020	42	4.38	440	csrss
83	5	892	860	31	0.11	3,268	csrss
192	11	12356	29136	99	5.30	772	dwm
182	8	16388	16204	76	0.17	3,244	dwm
1,370	55	32476	58200	343	20.00	3,688	explorer
0	0	0	28	0		0	Idle
345	15	8360	11136	167	0.53	1,180	LogonUI
947	17	3332	7256	35	3.98	540	lsass
440	17	50712	28376	172	92.78	1,464	MsMpEng
660	35	410748	438340	691	301.98	1,036	powershell
215	6	1580	4800	64	1.52	3,672	rdpclip
38	3	624	2600	32	0.03	2,580	rdrleakdiag
72	3	672	1832	31	0.08	996	RDVGHelper
82	4	680	700	32	0.08	1,820	RDVGHelper
562	18	16148	9168	102	1.66	3,060	SearchIndex...
242	7	3724	6732	40	4.23	532	services
39	1	180	472	3	0.05	280	smss
380	10	2972	5596	42	1.08	1,236	spoolsv

FIGURE 4-10 The *Out-GridView* cmdlet accepts pipelined input and displays a control that permits further exploration.

You can click the column headings to sort the output in descending order. Clicking the same column again changes the sort to ascending order. In Figure 4-11, the processes are sorted by the number of handles used by each process. The sort is from the largest number of handles to the smallest number of handles.

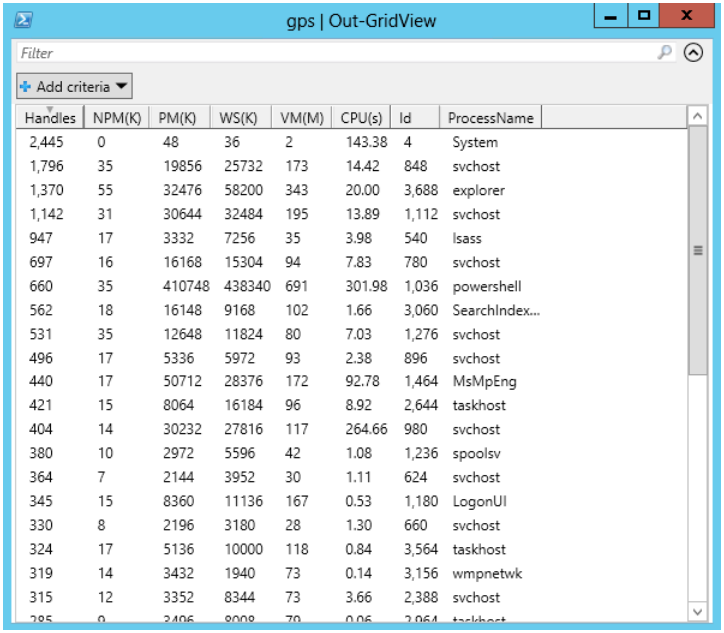


FIGURE 4-11 Clicking the column heading buttons permits sorting in either descending or ascending fashion.

The *Out-GridView* cmdlet accepts input from other cmdlets as well as from the *Get-Process* cmdlet. For example, you can pipeline the output from the *Get-Service* cmdlet to *Out-GridView* by using the syntax that appears here (*gsv* is an alias for the *Get-Service* cmdlet and *ogv* is an alias for the *Out-GridView* cmdlet).

```
gsv | ogv
```

Figure 4-12 shows the resulting Grid view.

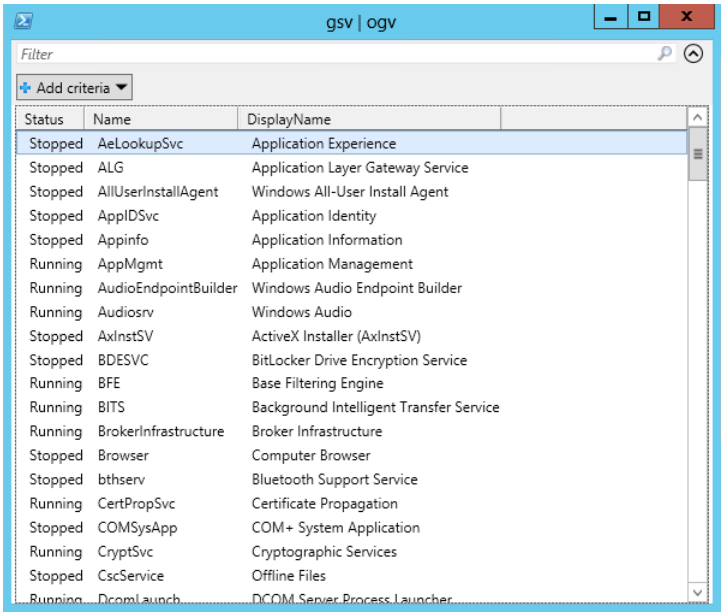


FIGURE 4-12 The *Out-GridView* cmdlet displays service controller information such as the current status of all defined services.

The *Out-GridView* cmdlet automatically detects the data type of the incoming properties. It uses this data type to determine how to present the filtering and the sorting information to you. For example, the data type of the Status property is a string. Clicking the Add Criteria button, choosing the Status property and selecting Add adds a filter that permits choosing various ways of interacting with the text stored in the Status property. The available options include the following: contains, does not contain, equals, does not equal, ends with, is empty, and is not empty. The options change depending upon the perceived data type of the incoming property.

Filtering output by using the filter box

To filter only running services, you can change the filter to "equals running." Keep in mind that if you choose an equality operator, your filtered string must match exactly. Therefore, "equals run" will not return any matches. Only "equals running" works. On the other hand, if you choose a "starts with" operator, you will find all the running services with the first letter. Therefore, "starts with r" returns everything. As you continue to type, matches continue to refine in the output.

TIP **Keep in the mind the difference in the behavior of the various filters. Depending on the operator you select, the self-updating output is extremely useful. This works especially well when attempting to filter out numerical data if you are not very familiar with the data ranges and what a typical value looks like. Figure 4-13 shows this technique.**

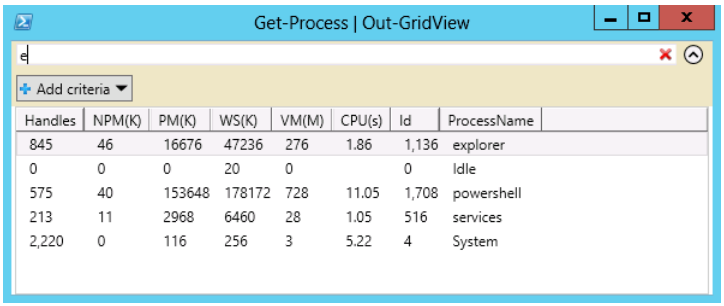


FIGURE 4-13 The *Out-GridView* self-updates when you type into the filter box.

By the time you type the first two letters of the explorer process name in the filter box, the resulting process information changes to display the single matching process name. Figure 4-14 shows the output.

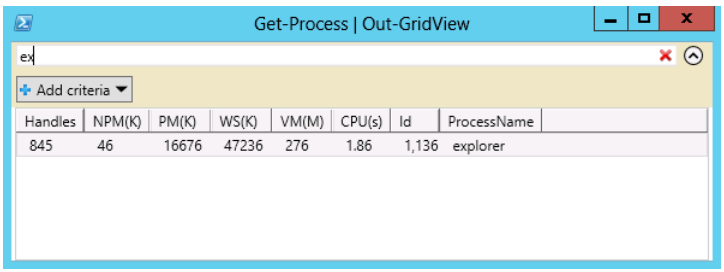


FIGURE 4-14 Clicking the red X at the end of the filter box clears the explore filter you added.

Summary

This chapter demonstrated how to use Windows PowerShell to create a table. Next, we discussed how to create a list and a wide display. We discussed the use of each of the three types of format as well as recommendations for most effective use. Finally, we examined the use of the output grid.

CHAPTER 5

Storing output

- Storing data in text files
- Storing data in .csv files
- Storing data in XML

When you work with Windows PowerShell in an interactive fashion from the Windows PowerShell console, there are times when you will want to store the output. On a Windows 8 computer, you can use the Snipping Tool to take a screen shot of the output. However, this is not an option on Windows Server 2012 unless you have installed the Desktop Experience feature. Of course, while having a screen shot does help you to have access to the data later, it does not facilitate parsing of the data.

Another option is copying to the Clipboard. With QuickEdit mode enabled for the Windows PowerShell console, you can use the mouse to highlight and copy text. You can then paste the text into any other program you want such as Notepad, Word, or even Outlook.

Storing data in text files

One of the easiest methods to store data is to store the data in a text file. In Figure 5-1, the output from the *Get-Volume* function displays in the Windows PowerShell console. The output formats nicely in columns and contains essential information about the volumes on a Windows 8 computer.

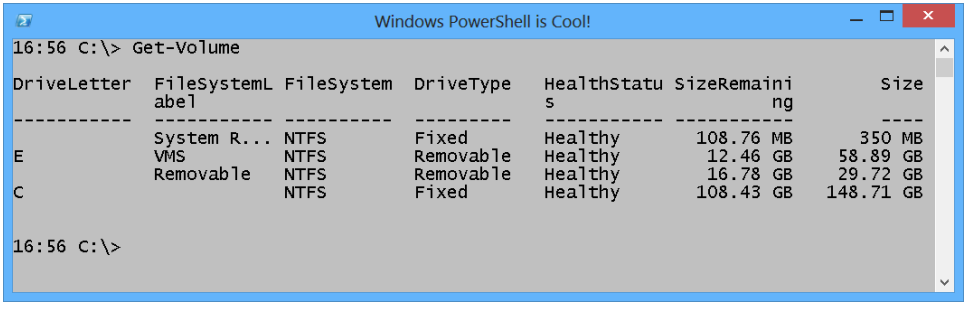


FIGURE 5-1 The *Get-Volume* function on Windows 8 displays essential information about the status of volumes.

Redirect and append

The easiest way to store volume information obtained from the *Get-Volume* function is to redirect the output to a text file. Because several lines of information return from the function, it is best to redirect and append the outputted information. The redirect and append operator is two right arrows, one behind the other with no space in between them (>>).

The following code redirects and appends the information from the *Get-Volume* function to a text file that resides in the folder c:\fso. The file, VolumeInfo.txt, might not exist. If it does not exist, it will be created and the information written to the file. If the file does exist, the outputted data will append to the file. The following example shows the command:

```
Get-Volume >>c:\fso\volumeinfo.txt
```

When the command runs, nothing outputs to the Windows PowerShell console. The output, formatted as it appears in the Windows PowerShell console, writes to the target text file. Figure 5-2 shows the Volumeinfo.txt file created by redirecting and appending the results of the *Get-Volume* function from Windows 8.

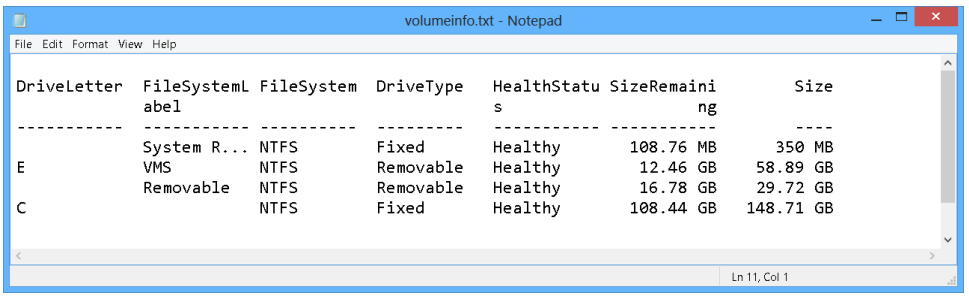


FIGURE 5-2 When you use the redirection and append operator, the output from the Windows PowerShell console writes to a text file for storage or documentation.

If you run the code that redirects and appends the information from the *Get-Volume* function to a text file named Volumeinfo.txt that resides in the folder c:\fso a second time, the information from *Get-Volume* writes to the bottom of the previously created text file; that is, it appends to the file. This is a great way to produce simple logging. Figure 5-3 shows the volume information appearing twice. In both cases, the values are identical. This shows that between the time the first *Get-Volume* command ran and the second time the *Get-Volume* ran, nothing changed.

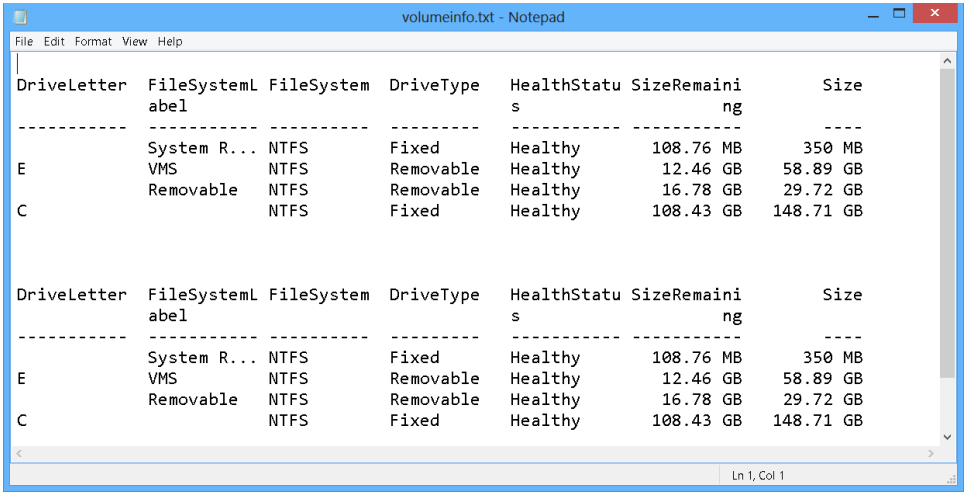


FIGURE 5-3 The redirect and append operator is great when you want to create a log file to check for changes over time.

Redirect and overwrite

If you do not need to maintain a history of prior settings, results, or data, use the redirect operator and do not append. The redirect and overwrite operator is a single right arrow (>).

The following code redirects and overwrites the information from the *Get-Volume* function to a text file that resides in the folder c:\fso. The file, VolumeInfo.txt, might not exist. If it does not exist, it will be created and the information written to the file. If the file does exist, the outputted data will overwrite previously existing data when writing to the file. The following example shows the command:

```
Get-Volume >c:\fso\volumeinfo.txt
```

Comparing the SizeRemaining value of the C drive from Figure 5-3 with the SizeRemaining value of the C drive in Figure 5-4 reveals that the drive suddenly has nearly 4 GB of additional free space. But because the Volumeinfo.txt file is overwritten by the redirect and overwrite operator, you have no way to discover this condition unless you have a backup of the previous Volumeinfo.txt file. Knowing when the 4 GB of disk space suddenly becomes available might assist the technical support agent when a user calls and says that "Outlook is not working."

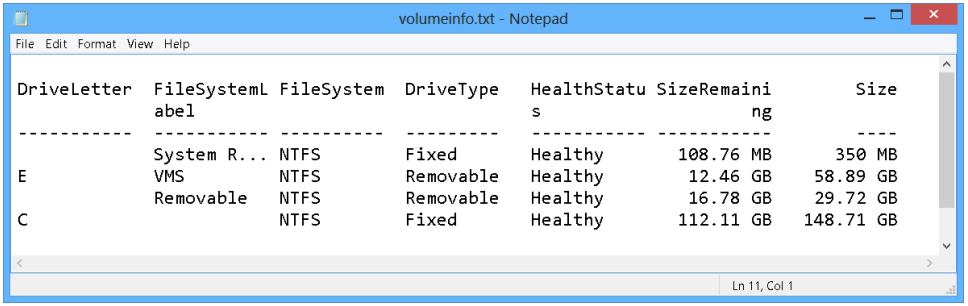


FIGURE 5-4 If you use the redirect operator without the append, previously existing files overwrite. This could mean a loss of data in many cases.

Controlling the text file

If you need to ensure your text file is a specific width (such as 152 columns wide) or a specific type of encoding (such as ASCII or Unicode), you will need to use the *Out-File* cmdlet. In addition to storing data, creating a very wide text file is a great way to overcome day-to-day space limitations of either the Windows PowerShell console or the Windows PowerShell ISE. In Figure 5-5, all the properties returned by the *Get-Service* cmdlet display to a table. Unfortunately, the computer screen resolution and the font size of the Windows PowerShell console do not permit displaying much information. In fact, nearly all the displayed information truncates to the point that the resulting output is unusable.

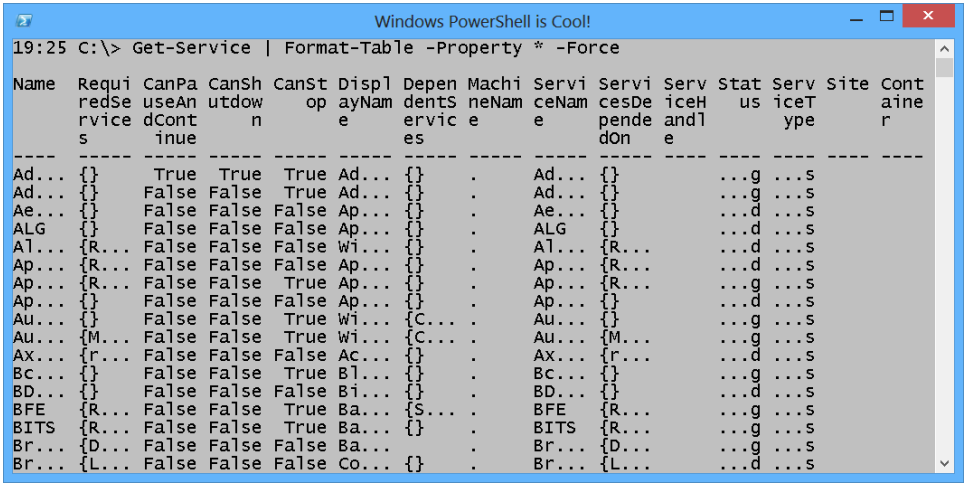


FIGURE 5-5 Screen resolution and font size in the Windows PowerShell console combine to make wide tables of data almost unusable.

When the same command that selects all properties from the *Get-Service* cmdlet and pipelines them to a table pipelines the results to a text file through the *Out-File* cmdlet instead of

writing to the Windows PowerShell console, the result is much more readable. To ensure that each row of data has enough space to write, increase the page width to 500. To ensure that Unicode characters are properly recorded, set the text encoding to UTF8.

> ***NOTE*** **Permissible encoding values for the *Out-File* cmdlet are: *string*, *unicode*, *bigendianunicode*, *utf8*, *utf7*, *utf32*, *ascii*.**

The following example shows the code to obtain all the service information to write it to a wide text file in table format:

```
Get-Service | Format-Table -Property * -Force -Auto |
Out-File -FilePath c:\fso\WideServices.txt -Encoding UTF8 -Width 500
```

Figure 5-6 shows the wide table in Notepad. The font size is reduced to permit more of the columns to appear on the page.

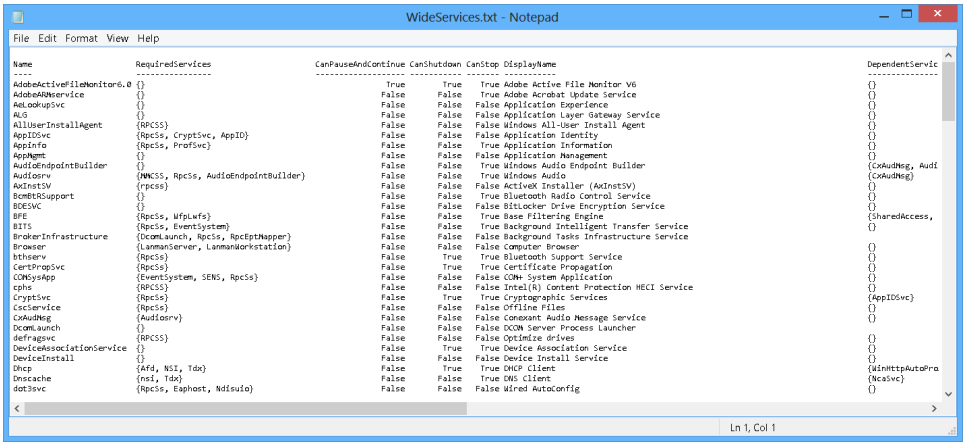


FIGURE 5-6 Avoid truncated columns by writing the wide table to a text file and specifying a wide width.

Storing data in .csv files

After a plain text file, the next level of complexity is a Comma Separated Value (.csv) file. Actually, by using the *Export-CSV* cmdlet, creating a .csv file is not very complicated.

No type information

The most important thing to remember when creating a .csv file is that if you want to open it in Microsoft Excel or import it to SQL or some other application, use the -NoTypeInformation switched parameter to avoid writing a line of type information to the top of the file. The following

example shows this technique of avoiding type information by using the NoTypeInformation switch when collecting process information:

```
Get-Process | Export-Csv -Path c:\fso\process.csv -NoTypeInformation
```

Opening the Process.csv text file in Notepad is not very illuminating. In fact, it is confusing due to the myriad commas and quotation marks. Figure 5-7 shows the Process.csv text file.

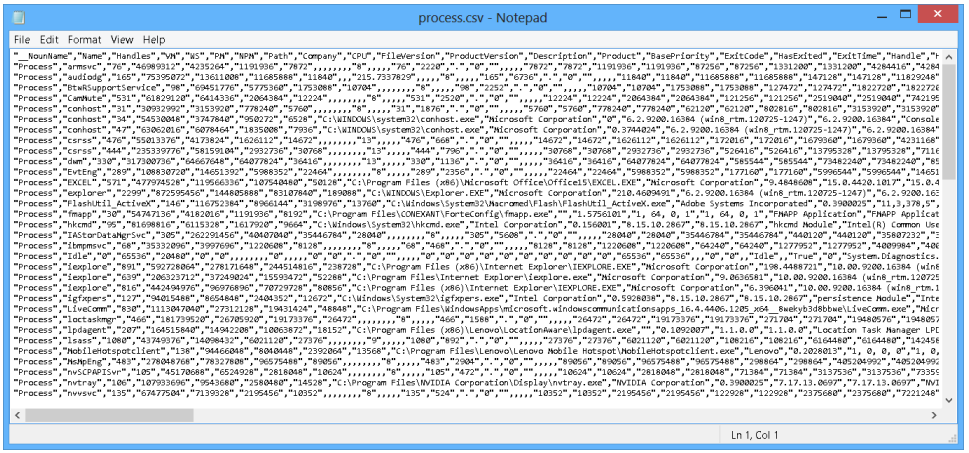


FIGURE 5-7 Opening a .csv file in Notepad is not the best use of the file format due to the numerous commas and quotation marks in the file.

A better way to use the .csv file format is to either import it to a database or open it in Microsoft Excel. When you open it in Microsoft Excel, you can easily manipulate the .csv file, sort the columns, and change the numbers to different formats. This ease of use makes pipelining data to a .csv file, and then opening and parsing in Microsoft Excel, a natural workflow. Figure 5-8 shows the same Process.csv file displayed in Figure 5-7 opened in Microsoft Excel. The difference in readability between the two figures is remarkable.

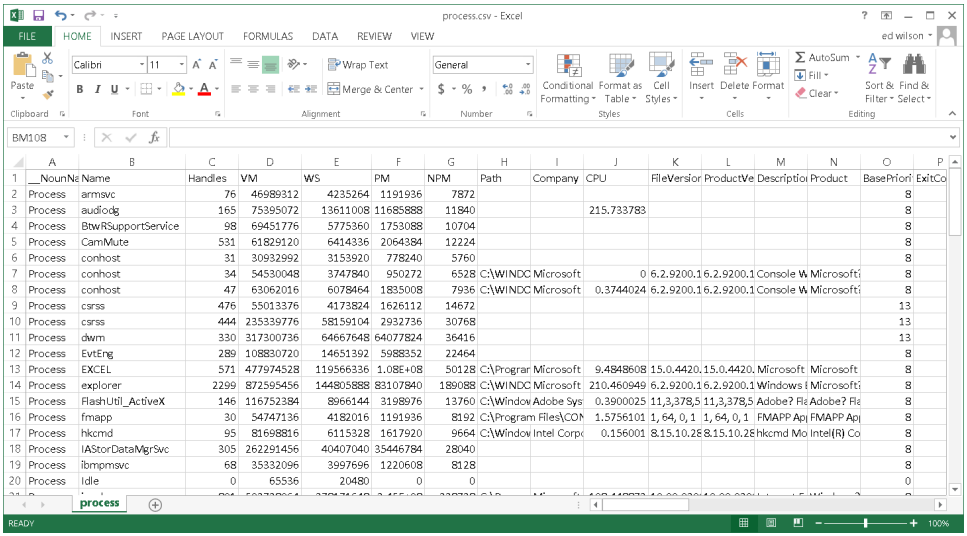


FIGURE 5-8 Opening a .csv file in Microsoft Excel is as easy as right-clicking on the file and selecting Open with Microsoft Excel.

Using type information

The way to use the type information, which is the same information that was removed in the "No type information" section of this chapter, is to provide information for reconstituting an object. It might sound confusing, but it actually is easy to do. You simply pipeline the results from a Windows PowerShell cmdlet to the *Export-CSV* cmdlet. This time you keep the type information. At a later point in time, you will import the CSV information from the .csv file, and you can use Windows PowerShell to analyze the data stored in the .csv file. This permits easy offline analysis of system data through Windows PowerShell and the .csv file.

> ***NOTE*** **Storing Windows PowerShell information into a .csv file for offline analysis is a great technique for consultants to use. You can have customers store the information in .csv files and email them to you so you can analyze the data and tell them what the problem is.**

The following example shows the offline analysis technique:

```
C:\> get-process | Export-Csv -Path c:\fso\processInfo.csv
C:\> Import-Csv -Path C:\fso\processInfo.csv | sort vm | select -First 2 |
ft name, vm

Name                                    VM
----                                    --
wlanext                                 100147200
wmpnetwk                                100188160
```

The first command stores all process information in a .csv file named ProcessInfo.csv. The *Export-CSV* cmdlet is used to perform the export. After the .csv file is created, the *Import-CSV* cmdlet imports the ProcessInfo.csv file. This command reconstitutes the process objects. The process objects are pipelined to the *Sort-Object* cmdlet (*sort* is an alias for *Sort-Object*) where the objects are sorted based upon the amount of virtual memory (VM) they consume. Next, the *Select-Object* cmdlet (*Select* is an alias for *Select-Object*) chooses the first two process objects. The first two process objects are pipelined to the *Format-Table* cmdlet (*ft* is an alias for *Format-Table*) where the name and the VM properties display.

Storing data in XML

Some objects are much more complex than the objects that could be stored easily in the relatively flat text format of a .csv file. Objects that are more complex are objects that have objects as values for their properties. For example, in the .csv file created in the previous section, the reconstituted objects do not contain any thread information. This is because thread information returned by the *Get-Process* cmdlet is a *ProcessThreadCollection* object.

The problem with complex objects

When you reconstitute an object from a .csv file, simple property/value pairs create perfectly. But objects stored in properties do not. The following example imports the CSV information from the ProcessInfo.csv file and stores the recreated objects in a variable named *$csv*:

```
C:\> $csv = Import-Csv -Path C:\fso\processInfo.csv
```

The following example retrieves the name of the first object. This code works fine:

```
C:\> $csv[0].name
Armsvc
```

Now, the VM property of the Armsvc process is retrieved. This code works as well:

```
C:\> $csv[0].vm
46989312
```

However, when you are attempting to retrieve the threads of the *Armsvc* process, a string stating that it is a *System.Diagnostics.ProcessThreadCollection* object will return:

```
C:\> $csv[0].threads
System.Diagnostics.ProcessThreadCollection
```

Using XML to store complex objects

The solution to storing an offline representation of a complex object is to use XML. Because Windows PowerShell handles the process of creating the XML and interpreting the XML, you do not need to know anything about XML. This is great because essentially you get the power

of using XML without having to deal with any of the complexities of XML. To create an XML representation of the object, pipeline the results from the cmdlet to the *Export-Clixml* cmdlet and specify the path. Just like the *Export-CSV* cmdlet, there is a -NoTypeInformation parameter, but unlike CSV, there are not too many easy-to-use applications that facilitate perusing an offline .xml file. The following example shows the command to create the .xml file:

```
Get-Process | Export-Clixml -Path c:\fso\processXML.xml
```

Now, it is certainly possible to open the .xml file and peruse it. Figure 5-9 shows the .xml file as it appears in Internet Explorer.

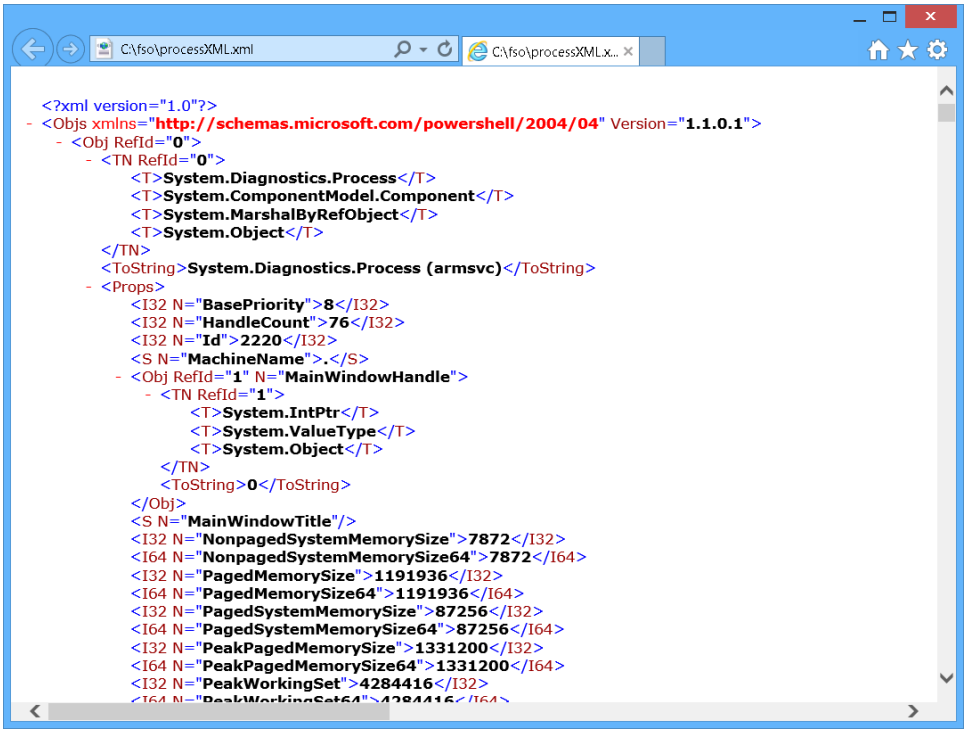


FIGURE 5-9 It is possible to open an exported .xml file in another application, but generally it is easier to reconstitute the object and use Windows PowerShell to parse the data.

To reconstitute the object from the offline .xml file, use the *Import-Clixml* cmdlet and store the objects in a variable:

```
C:\> $xml = Import-Clixml -Path C:\fso\processXML.xml
```

To view the name of the first process object, choose the *Name* property from the first element:

```
C:\> $xml[0].name
Armsvc
```

To see the amount of virtual memory used by the Armsvc process, use the *VM* property:

```
C:\> $xml[0].vm
46989312
```

Now, to see the threads used by the Armsvc process, all that is required is to access the *Threads* property:

```
C:\> $xml[0].threads


BasePriority    : 8
CurrentPriority : 9
Id              : 2224
StartAddress    : 8776924411104
ThreadState     : Wait
WaitReason      : UserRequest
Site            :
Container       :

BasePriority    : 8
CurrentPriority : 8
Id              : 2232
StartAddress    : 8776924411104
ThreadState     : Wait
WaitReason      : EventPairLow
Site            :
Container       :

BasePriority    : 8
CurrentPriority : 9
Id              : 2240
StartAddress    : 8776924411104
ThreadState     : Wait
WaitReason      : UserRequest
Site            :
Container       :
```

Summary

This chapter examined storing output from Windows PowerShell cmdlets. In particular, we covered creating a text file to store the results of commands. We can use Windows PowerShell to facilitate bringing data into other tools such as a database or a spreadsheet.

CHAPTER 6

Leveraging Windows PowerShell providers

- Understanding Windows PowerShell providers
- Understanding the Alias provider
- Understanding the Certificate provider
- Understanding the Environment provider
- Understanding the File System provider
- Understanding the Function provider
- Understanding the Registry provider
- Understanding the Variable provider

One of the most important concepts with Windows PowerShell is the concept of Windows PowerShell providers. Windows PowerShell providers permit you to use the same cmdlets, such as *Get-Item* or *Set-Item*, to work with different types of data. This allows you to know immediately how to work with lots of different types of data. For example, you can use the *Get-Item* cmdlet to retrieve information about a file. However, you can use the same cmdlet to retrieve information about an alias, a certificate, a function, an environment variable, a registry key, or a variable. The same cmdlet with the same basic parameters used in the same way permits you to work with many different types of data. In addition, Windows PowerShell providers are extensible; that is, other teams at Microsoft and third-party vendors can write additional providers to provide access to their data stores. The Microsoft Active Directory provider permits you to use cmdlets like *Get-Item* to retrieve information about a user, a computer, or other object in Microsoft Active Directory Domain Services (AD DS). In addition, an SQL provider exposes information about Microsoft SQL Server databases. In the community, there are XML providers that permit working with XML documents, and there is an SQLite provider for working with an SQLite database. In this chapter, we examine the Alias, Certificate, Environment, File System, Function, Registry, and Variable providers.

Understanding Windows PowerShell providers

By identifying the providers installed with Windows PowerShell, we can begin to understand the capabilities intrinsic to a default installation. Providers expose information contained in different data stores by using a drive and file system analogy. An example of this is obtaining a listing of registry keys. To do this, you connect to the registry drive and use the *Get-ChildItem* cmdlet, which is the same method you use to obtain a listing of files on the hard drive. The only difference is the specific name associated with each drive.

To obtain a listing of all providers, use the *Get-PSProvider* cmdlet. This command produces the following list on a default installation of Windows PowerShell:

```
PS C:\> Get-PSProvider

Name                 Capabilities                    Drives
----                 ------------                    ------
Alias                ShouldProcess                   {Alias}
Environment          ShouldProcess                   {Env}
FileSystem           Filter, ShouldProcess, Crede... {C}
Function             ShouldProcess                   {Function}
Registry             ShouldProcess, Transactions     {HKLM, HKCU, HKCR}
Variable             ShouldProcess                   {Variable}
```

Understanding the Alias provider

In Chapter 1, "Overview of Windows PowerShell 3.0," we examined the various Help utilities available that show how to use cmdlets. The alias provider provides easy-to-use access to all aliases defined in Windows PowerShell. An alias is a shortcut name for a cmdlet or for a function. Aliases make it easier to work interactively with Windows PowerShell from within the Windows PowerShell console. Because aliases are customizable, you can create your own aliases for existing cmdlets and functions or for new cmdlets or functions that you write in the future. If you do not like an existing alias, you can delete it or you can change its meaning.

> ***NOTE*** **Because of the dynamic nature of aliases, it is not wise to use an alias in a Windows PowerShell script that you intend to share with others. If you are the only one who will ever use a particular Windows PowerShell script, then it is fine to use an alias because you know the alias will exist only on your system. But you can never be sure if an alias will exist on someone else's computer, and therefore as a best practice you should avoid using aliases in Windows PowerShell scripts you intend to share.**

To work with the aliases on your machine, use the *Set-Location* cmdlet and specify the Alias:\ drive. You can then use the same cmdlets you would use to work with the file system. These include the *Get-Item*, *Set-Item*, *New-Item*, and *Remove-Item* cmdlets.

TIP **With the Alias provider, you can use a *Where-Object* cmdlet and filter to search for an alias by name or description.**

The following example creates a new alias named *Processes* for the *Get-Process* cmdlet:

```
PS C:\> Set-Location alias:
PS Alias:\> New-Item -Name Processes -Value Get-Process

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           Processes -> Get-Process
```

To use the newly created alias, just type the alias name at the Windows PowerShell command prompt. As shown in the following example, the newly created *Processes* alias returns the first process:

```
PS Alias:\> processes | select -First 1

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     76       8     1156       4096    45            2220 armsvc
```

To delete the alias, use the *Remove-Item* cmdlet. Because you are still using the Alias:\ drive, it is unnecessary to supply the complete path to the alias. In fact, using tab expansion causes .\ to prepend to the alias name. This is because .\ refers to the current directory. Nothing returns to the Windows PowerShell console when the *Remove-Item* cmdlet runs, but if the item does not exist, an error appears:

```
PS Alias:\> Remove-Item .\Processes
```

If you want to verify the removal of an alias, you can use the *Get-Item* cmdlet. When you do this, an error appears in the Windows PowerShell console because the alias *Processes* was already deleted in the previous command. The following example shows this verification process:

```
PS Alias:\> Get-Item .\processes
Get-Item : Cannot find path 'Alias:\processes' because it does not exist.
At line:1 char:1
+ Get-Item .\processes
+ ~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (Alias:\processes:String) [Get-Item],
    ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCom
   Mand
```

You do not need to use the Alias provider and the *Item cmdlets, such as *Get-Item* and *New-Item*, to work with aliases because there are five cmdlets designed specifically to work with Windows PowerShell aliases. The following example shows these cmdlets:

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

Name
----
Export-Alias
Get-Alias
Import-Alias
New-Alias
Set-Alias
```

For example, to create a new alias by using the *New-Alias* cmdlet, you need to specify the name for the alias and the value. These correspond with the *New-Item* cmdlet parameters. The big difference is that the path is not required because *New-Alias* automatically creates the newly created alias in the proper location. The following example shows the command:

```
PS C:\> New-Alias -Name myservice -Value Get-Service
```

Once created, the newly created alias works just like any other alias, as shown in the following example:

```
PS C:\> myservice | select name -First 1

Name
----
AdobeActiveFileMonitor6.0
```

There is no *Remove-Alias* cmdlet, so you must use the *Remove-Item* cmdlet to remove an alias and specify the complete path to the alias. The following example shows this technique from the C drive:

```
PS C:\> Remove-Item -Path Alias:\myservice
```

This requirement for the path prevents you from pipelining the results of *Get-Alias* to the *Remove-Item* cmdlet. To pipeline an alias to the *Remove-Item* cmdlet, you must use the Alias drive, as shown in the following example:

```
PS C:\> New-Alias -Name sample -Value Get-EventLog
PS C:\> Get-Item Alias:\sample | Remove-Item
```

Understanding the Certificate provider

In the preceding section, we explored working with the Alias provider. Because the file system model applies to the Certificate provider in much the same way as it did the Alias provider, you can use many of the same cmdlets. To find information about the Certificate provider, use the *Get-Help* cmdlet. If you are unsure what topics in Help may be related to certificates, you can use the wild card asterisk (*) parameter:

```
Get-Help *cer*
```

The Certificate provider gives you the ability to sign scripts and allows Windows PowerShell to work with signed and unsigned scripts as well. It also gives you the ability to search for, copy, move, and delete certificates. Using the certificate provider, you can even open the Certificates Microsoft Management Console (MMC) by using the *Invoke-Item* cmdlet. The following example illustrates this technique:

```
Invoke-Item cert:
```

> ***NOTE*** **The Certificate provider does not load by default. The module that contains the Certificate provider, *Microsoft.PowerShell.Security*, does not automatically import into every session. To use the Cert drive, use the *Import-Module* cmdlet to import the module or run a command that uses the Cert drive, such as a "*Set-Location Cert*" command.**

Searching for specific certificates

To search for specific certificates, you might want to examine the Subject property. For example, the following code examines the Subject property of every certificate in the currentuser store beginning at the root level. It does a recursive search and returns only the certificates that contain the word test in some form in the Subject property. The following example shows the command and associated output:

```
PS C:\Users\administrator.IAMMRED> dir Cert:\CurrentUser -Recurse | ? subject -match
'test'

    Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\Root


Thumbprint                                Subject
----------                                -------
8A334AA8052DD244A647306A76B8178FA215F344  CN=Microsoft Testing Root Certificate A...
2BD63D28D7BCD0E251195AEB519243C13142EBC3  CN=Microsoft Test Root Authority, OU=Mi...
```

To delete these test certificates, you simply need to pipeline the results of the previous command to the *Remove-Item* cmdlet.

> ***IMPORTANT*** **When you perform any operation that might alter system state, it is a good idea to use the Whatif parameter to prototype the command prior to actually executing it.**

The following command uses the Whatif parameter from *Remove-Item* to prototype the command to remove all the certificates from the *currentuser* store that contain the word *test* in the Subject property. Once completed, retrieve the command by pressing the Up arrow key and removing the Whatif switched parameter from the command prior to actual execution. The following example shows this technique:

```
PS C:\Users\administrator.IAMMRED> dir Cert:\CurrentUser -Recurse | ? subject -match
'test' | Remove-Item -WhatIf
```

```
What if: Performing operation "Remove certificate" on Target "Item: CurrentUser\Root\
8A334AA8052DD244A647306A76B8178FA215F344 ".
What if: Performing operation "Remove certificate" on Target "Item: CurrentUser\Root\
2BD63D28D7BCD0E251195AEB519243C13142EBC3 ".
PS C:\Users\administrator.IAMMRED> dir Cert:\CurrentUser -Recurse | ? subject -match
'test' | Remove-Item
```

Finding expiring certificates

A common task in companies using certificates is to identify certificates that have either expired or are about to expire. Using the Certificate provider, it is simple to identify expiring or expired certificates. To do this, use the NotAfter property from the certificate objects returned from the certificate drives. One approach is to look for certificates that expire prior to a specific date:

```
PS Cert:\> dir .\\CurrentUser -Recurse | where notafter -lt "5/1/2012"
```

A more flexible approach is to use the current date because each time the command runs it retrieves expired certificates:

```
PS Cert:\> dir .\\CurrentUser -Recurse | where notafter -lt (Get-Date)
```

One problem with simply using the *Get-ChildItem* cmdlet on the *currentuser* store is that it returns both certificate stores as well as certificates. To obtain only certificates, you must filter out the PSISContainer property. Because you will also need to filter based on date, it means you no longer can use the simple *Where-Object* syntax.

The following command retrieves the expiration dates, the thumbprints, and the subjects of all expired certificates. It also creates a table displaying the information. The command is a single logical command, but it is broken at the pipeline character to permit better display in the book:

```
PS Cert:\> dir .\\CurrentUser -Recurse |
where { !$_.psiscontainer -AND $_.notafter -lt (Get-Date)}  |
ft notafter, thumbprint, subject -AutoSize -Wrap
```

> ***WARNING*** **All versions of Microsoft Windows ship with expired certificates to permit verification of old executables that were signed with those certificates. Do not arbitrarily delete an expired certificate, or you could cause serious damage to your system.**

If you want to identify certificates that will expire in the next 30 days, use the dynamic parameter -ExpiringInDays from the *Get-ChildItem* cmdlet. This dynamic parameter adds to the *Get-ChildItem* cmdlet when it is used on the Cert drive:

```
PS Cert:\> Get-ChildItem -Recurse -ExpiringInDays 30
```

To produce a useful display, select the Subject and the NotAfter parameters and sort by the NotAfter parameter. Then pipeline the output to a table that is autosized and wrapped. The following example shows the command:

```
PS Cert:\> gci -ExpiringInDays 30 -r | select subject, notafter | sort notafter | ft
notafter, subject -a -wr

NotAfter             Subject
--------             -------
2/12/2013 6:34:47 PM
2/16/2013 2:56:37 PM CN=KenMyer@microsoft.com
3/4/2013 4:42:09 PM  CN=Microsoft Corporation, OU=MOPR, O=Microsoft Corporation,
                     L=Redmond, S=Washington, C=US
3/4/2013 4:42:09 PM  CN=Microsoft Corporation, OU=MOPR, O=Microsoft Corporation,
                     L=Redmond, S=Washington, C=US
```

Understanding the Environment provider

The Environment provider in Windows PowerShell provides access to the system environment variables. If you open a Command Prompt window and type set, you will obtain a listing of all environment variables defined on the system. (You can run the old-fashioned command prompt inside Windows PowerShell.)

> ***NOTE*** **It is easy to forget you are running the command-line program when you use the Windows PowerShell console. The best way to determine if you are running the command-line program or Windows PowerShell is to examine the prompt. The default Windows PowerShell prompt is PS C:\>, assuming you are working on the C drive.**

Figure 6-1 shows the results if you use the *echo* command in the command interpreter to print the value of *%windir%*.

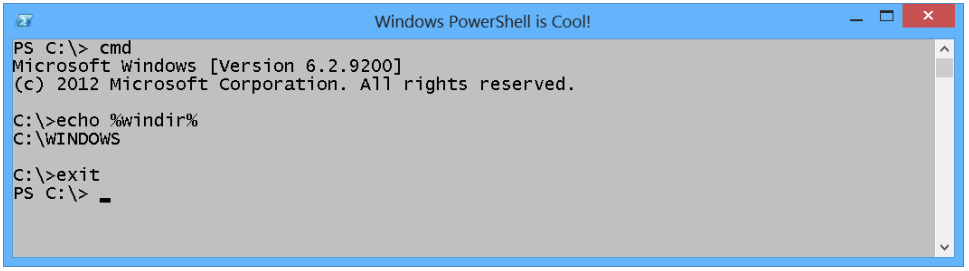


FIGURE 6-1 Use *echo* in a command prompt to see the value of an environmental variable.

Various applications and other utilities use environment variables as a shortcut to provide easy access to specific files, folders, and configuration data. By using the Environment provider in Windows PowerShell, you can obtain a listing of the environment variables. You can also add, change, clear, and delete these variables.

To obtain a listing of all environmental variables, use the *Get-ChildItem* cmdlet on the Env drive (you can also use the alias *dir*, *ls,* or even *gci* to do this). The following example shows this technique:

```
PS C:\> Set-Location env:
PS Env:\> dir

Name                           Value
----                           -----
ALLUSERSPROFILE                C:\ProgramData
APPDATA                        C:\Users\ed.IAMMRED\AppData\Roaming
CommonProgramFiles             C:\Program Files\Common Files
CommonProgramFiles(x86)        C:\Program Files (x86)\Common Files
<… Output truncated …>
```

To display the value of a specific environmental variable, you can use the *Get-Item* cmdlet:

```
PS Env:\> Get-Item windir

Name                           Value
----                           -----
windir                         C:\WINDOWS
```

This technique is great if you want to see the environmental variable as well as the value associated with the variable. But if you want only the value, perhaps because you want to use the value somewhere else, it gets a little tricky. One way to do this is to use the group and dot methodology in which you group the command and then access the specific property, as shown in the following example:

```
PS Env:\> (Get-Item windir).value
C:\WINDOWS
```

A more direct way uses a shortcut to access the value directly:

```
PS C:\> $env:windir
C:\WINDOWS
```

Understanding the File System provider

The File System provider is the easiest Windows PowerShell provider to understand because it simply provides access to the file system. When Windows PowerShell is launched, it automatically opens on the C: Powershell Drive. Using the Windows PowerShell File System provider, you can create both directories and files. You can retrieve properties of files and directories, and you can delete them as well. In addition, you can open files and append or overwrite data to the files. This can be done with inline code or by using the pipelining feature of Windows PowerShell.

To create a new folder, use the *New-Item* cmdlet and specify the path for the new folder to reside. You will also need to specify the *ItemType* of the directory:

```
PS C:\> New-Item -Path C:\samplefolder -ItemType directory


    Directory: C:\


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----          2/9/2013   9:26 PM            samplefolder
```

If you do not want the returned *DirectoryInfo* object when you create the new folder, pipeline the results to the *Out-Null* cmdlet:

```
PS C:\> New-Item -Path C:\samplefolder\sub1 -ItemType directory | Out-Null
PS C:\>
```

Use the *Set-Location* cmdlet (*cd* is an alias) to change to the newly created folder. After you open the folder, you can use *New-Item* to create a new file. Because you changed the working location to the newly created directory, it is unnecessary to specify the path for the new file. You need to specify only the name. The following example shows this technique:

```
PS C:\> Set-Location C:\samplefolder
PS C:\samplefolder> New-Item -Name samplefile.txt -ItemType file


    DirectorY: C:\samplefolder


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          2/9/2013   9:28 PM          0 samplefile.txt
```

To write to the newly created file, use the *Add-Content* cmdlet. The required parameters for the *Add-Content* cmdlet are the path to the file and the value to write to the file:

```
PS C:\samplefolder> Add-Content -Path .\samplefile.txt -Value "this is new content"
```

To read the contents of a file, use the *Get-Content* cmdlet:

```
PS C:\samplefolder> Get-Content -Path .\samplefile.txt
this is new content
```

Instead of using the *Set-Location* cmdlet, you can use the alias *CD* to change your working location on the drive. For example, to change from C:\samplefolder back to the root of the drive, you can use the following technique:

```
PS C:\samplefolder> cd C:\

PS C:\>
```

To delete a folder, use the *Remove-Item* cmdlet. If the folder has additional folders or files inside it, you need to use the -Recurse parameter. If you do not specify -Recurse, a command prompt appears. The following example shows this technique:

```
PS C:\> Remove-Item C:\samplefolder

Confirm
The item at C:\samplefolder has children and the Recurse parameter was not
specified. If you continue, all children will be removed with the item. Are you sure
 you want to continue?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y
```

Understanding the Function provider

The Function provider provides access to the functions defined in Windows PowerShell. By using the Function provider, you can obtain a listing of all functions on your system. You can also add, modify, and delete functions. The Function provider uses a file system–based model, and the cmdlets you learned earlier apply to working with functions.

Use the *Get-ChildItem* cmdlet to obtain a listing of all the functions defined on the system:

```
PS C:\> Get-ChildItem function:

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        A:
Function        Add-HeaderToScript                                 PowerShellISEM..
Function        Add-Help                                           PowerShellISEM..
Function        Add-SBSBookHeaderToScript                          PowerShellISEM..
Function        B:
<… Output Truncated …>
```

To see the contents of a function, use the *Get-Content* cmdlet to read the contents of the function from the function drive. The following example reads the contents of the *Prompt* function:

```
PS C:\> Get-Content Function:\prompt
"PS $($executionContext.SessionState.Path.CurrentLocation)$('>' *
($nestedPromptLevel + 1)) "
# .Link
# http://go.microsoft.com/fwlink/?LinkID=225750
# .ExternalHelp System.Management.Automation.dll-help.xml
```

To modify a system function, you might want to create a backup of the function first. To do this, redirect the output to a file:

```
PS C:\> Get-Content Function:\prompt >>C:\fso\PromptFunction.txt
```

Understanding the Registry provider

In Windows PowerShell 1.0, the Registry provider makes it easy to work with the registry on the local system. Unfortunately, without remoting you are limited to working with the local computer or using some other remoting mechanism such as a logon script to make changes on remote systems. Beginning with Windows PowerShell 2.0, the inclusion of remoting makes it possible to make remote registry changes as easily as changing the local registry.

> ***CAUTION*** **The registry contains information vital to the operation and configuration of your computer. Serious problems could arise if you edit the registry incorrectly. Therefore, it is important to back up your system prior to attempting to make any changes. For information about backing up your registry, see KB322756. For general information about working with the registry, see KB310516 and KB256986.**

The Registry provider permits access to the registry in the same manner that the File System provider permits access to a local disk drive. The same cmdlets that used to access the file system—such as *New-Item*, *Get-ChildItem*, *Set-Item*, and *Remove-Item*—also work with the registry. In addition to these cmdlets, if you want to work with a specific registry item value, you might need to use *New-ItemProperty*, *Get-ItemProperty*, *Set-ItemProperty*, and *Remove-ItemProperty*.

The two registry drives

By default, the Registry provider creates two registry drives. To find all the drives exposed by the Registry provider, use the *Get-PSDrive* cmdlet. The following example shows these drives:

```
PS C:\> Get-PSDrive -PSProvider registry | select name, root

Name                                   Root
----                                   ----
HKCU                                   HKEY_CURRENT_USER
HKLM                                   HKEY_LOCAL_MACHINE
```

You can create additional registry drives by using the *New-PSDrive* cmdlet. For example, it is common to create a registry drive for the HKEY_CLASSES_ROOT registry hive, as shown in the following example:

```
PS C:\> New-PSDrive -PSProvider registry -Root HKEY_CLASSES_ROOT -Name HKCR

WARNING: column "CurrentLocation" does not fit into the display and was removed.

Name           Used (GB)     Free (GB) Provider      Root
----           ---------     --------- --------      ----
HKCR                                   Registry      HKEY_CLASSES_ROOT
```

Once created, the new HKCR drive is accessible in the same way as any other drive. For example, to change the working location to the HKCR drive, use either the *Set-Location* cmdlet or one of its aliases (such as *cd*):

```
PS C:\> Set-Location HKCR:
```

To determine the current location, use the *Get-Location* cmdlet:

```
PS HKCR:\> Get-Location

Path
----
HKCR:\
```

Once set, explore the new working location by using the *Get-ChildItem* cmdlet (or one of the aliases for that cmdlet such as *dir*). Figure 6-2 shows this technique.

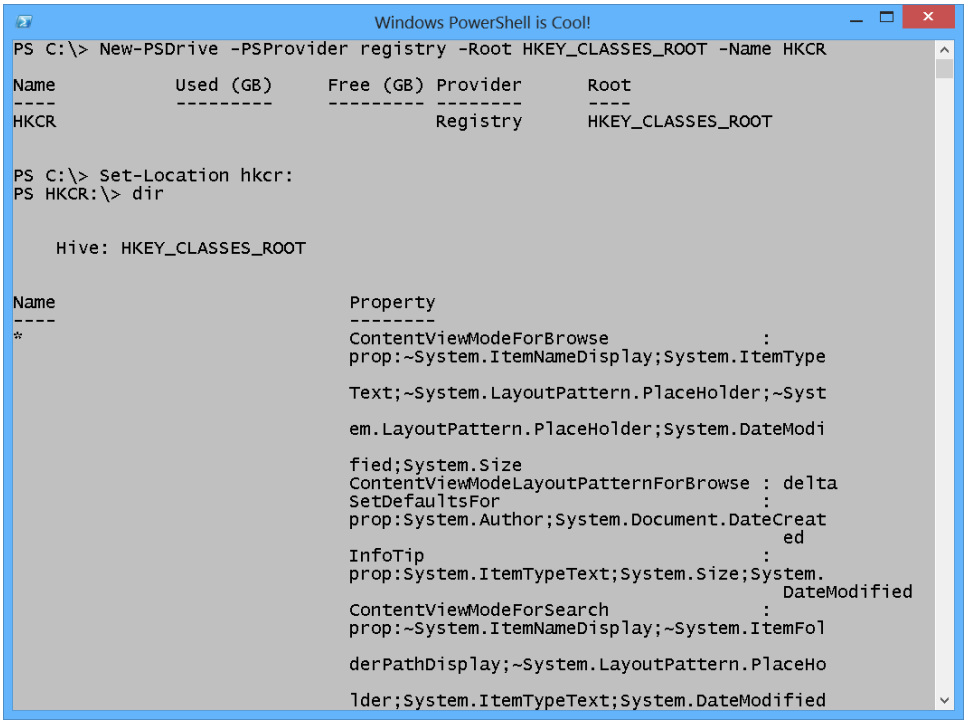


FIGURE 6-2 Creating a new registry drive for the HKEY_CLASSES_ROOT registry hive enables easy access to class registration information.

Retrieving registry values

To view the values stored in a registry key, use either the *Get-Item* or the *Get-ItemProperty* cmdlet. Using the *Get-Item* cmdlet reveals there is one property (named *default*), as shown in the following example:

```
PS HKCR:\> Get-Item .\.ps1 | fl *


PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT\.ps1
PSParentPath  : Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT
PSChildName   : .ps1
PSDrive       : HKCR
PSProvider    : Microsoft.PowerShell.Core\Registry
PSIsContainer : True
Property      : {(default)}
SubKeyCount   : 1
ValueCount    : 1
Name          : HKEY_CLASSES_ROOT\.ps1
```

To access the value of the default property, you must use the *Get-ItemProperty* cmdlet:

```
PS HKCR:\> Get-ItemProperty .\.ps1 | fl *

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT\.ps1
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT
PSChildName  : .ps1
PSDrive      : HKCR
PSProvider   : Microsoft.PowerShell.Core\Registry
(default)    : Microsoft.PowerShellScript.1
```

Figure 6-3 shows the technique for accessing registry keys and the values associated with them.

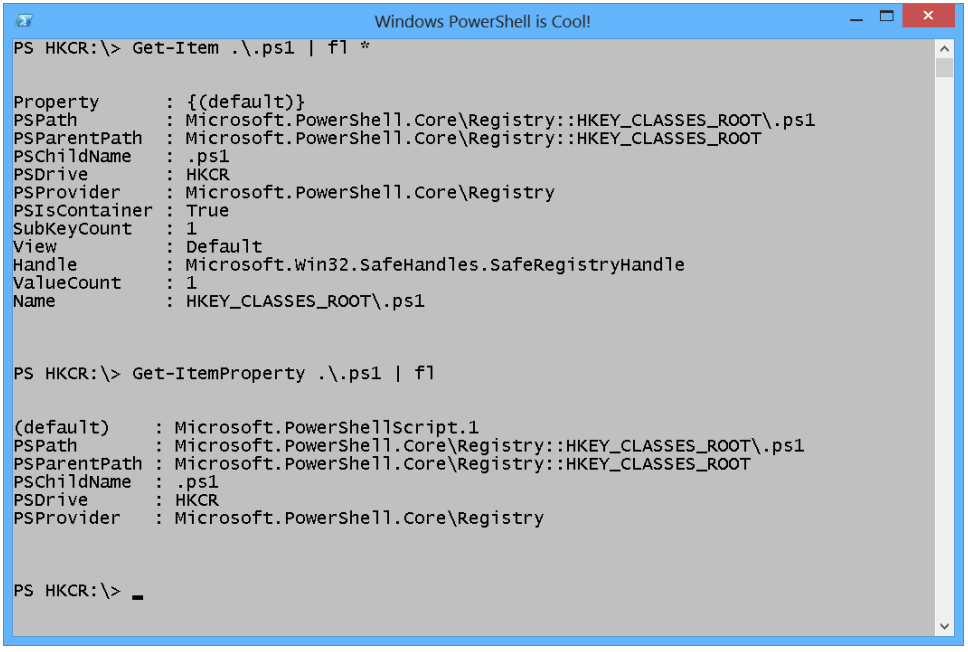


FIGURE 6-3 Use the *Get-ItemProperty* cmdlet to access registry property values.

To return only the value of the default property requires a bit of manipulation. The default property requires using literal quotation marks to force the evaluation of the parentheses in the name, as shown in the following example:

```
PS HKCR:\> (Get-ItemProperty .\.ps1 -Name '(default)').'(default)'
Microsoft.PowerShellScript.1
PS HKCR:\>
```

Two *PSDrives* are created by default. To identify the *PSDrives* that are supplied by the Registry provider, you can use the *Get-PSDrive* cmdlet, pipeline the resulting objects into the *Where-Object* cmdlet, and filter on the *provider* property while supplying a value that is like the word *registry*. The following example shows this command:

```
Get-PSDrive | where Provider -like "*Registry*"
The resulting list of PSDrives is shown here:
Name       Provider      Root                                    CurrentLocation
----       --------      ----                                    ---------------
HKCU       Registry      HKEY_CURRENT_USER
HKLM       Registry      HKEY_LOCAL_MACHINE
```

Creating new registry keys

Creating a new registry key by using Windows PowerShell is the same as creating a new file or a new folder because all three processes use the *New-Item* cmdlet. In addition, you could use the *Test-Path* cmdlet to determine if the registry key already exists. You might also want to change your working location to one of the registry drives. If you do this, you could use the *Push-Location*, *Set-Location,* and *Pop-Location* cmdlets. This is, of course, the long way of doing things.

The following example creates a new registry key named HSG off the HKEY_CURRENT_USERS software registry hive. It illustrates the five cmdlets mentioned in the preceding paragraph:

```
Push-Location
Set-Location HKCU:
Test-Path .\Software\sample
New-Item -Path .\Software -Name sample
Pop-Location
```

Figure 6-4 shows the commands and the associated output.

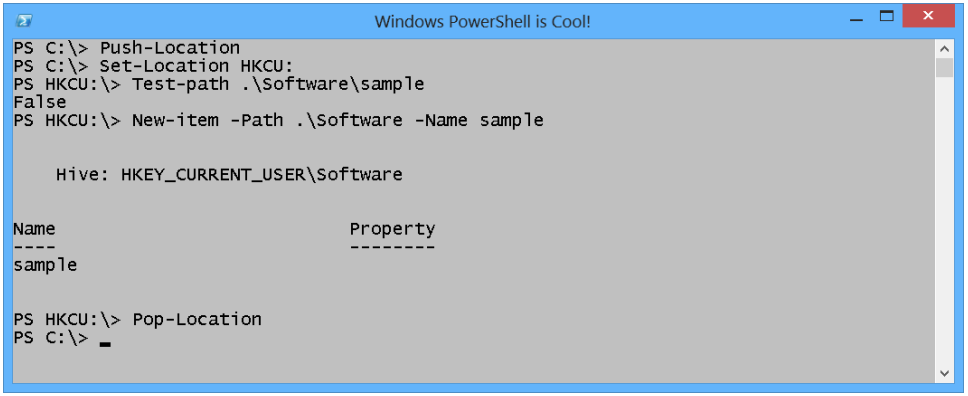


FIGURE 6-4 Creating a new registry key by using the *New-Item* cmdlet.

The short way to create a new registry key

It is not always necessary to change the working location to a registry drive when you create a new registry key. In fact, it is not even necessary to use the *Test-Path* cmdlet to determine if the registry key exists. If the registry key already exists, an error generates. If you want to overwrite the registry key, use the Force parameter.

> ***NOTE*** **IT pros who venture very far into the world of scripting need to make a design decision about how to deal with an already existing registry key. Software developers are very familiar with these types of decisions and usually deal with them in the analyzing requirements stages of the development lifecycle. IT pros who open the Windows PowerShell ISE first and think about the design requirements second become easily stymied or else write in problems. For more information, see my Microsoft Press book, *Windows PowerShell 2.0 Best Practices*.**

The following example creates a new registry key named Test in the HKCU:\Software location:

```
New-Item -Path HKCU:\Software -Name test -Force
```

Because the command includes the full path, it does not need to execute from the HKCU drive. The command uses the Force switched parameter, so the command overwrites the HKCU:\Software\test registry key if it already exists.

Setting the default value for the key

The preceding examples do not set the default value for the newly created registry key. If the registry key already exists (as it does in this specific case) use the *Set-Item* cmdlet to assign a default value to the registry key. To accomplish this, use the *Set-Item* cmdlet and supply the complete path to the existing registry key. Next, supply the default value in the Value parameter of the *Set-Item* cmdlet.

The following command assigns the value "*sample key*" to the default property value of the sample registry key contained in the HKCU:\Software location:

```
Set-Item -Path HKCU:\Software\sample -Value "sample key"
```

Using *New-Item* to create and assign a value

It is unnecessary to use the *New-Item* cmdlet to create a registry key and then use the *Set-Item* cmdlet to assign a default value. You can combine these into a single command. The following command creates a new registry key with the name Sample and assigns a default value of "*default value*" to the registry key:

```
New-Item -Path HKCU:\Software\sample -Value "default value"
```

Modifying the value of a registry property value

To modify the value of a registry property value, you must use the *Set-PropertyItem* cmdlet. To begin changing the registry property value, use the *Push-Location* cmdlet to save the current working location. Next, use the *Set-Location* cmdlet to change to the appropriate registry drive, and then use the *Set-ItemProperty* cmdlet to assign a new value to the registry property. Finally, use the *Pop-Location* cmdlet to return to the original working location. The following example shows this technique:

```
PS C:\> Push-Location
PS C:\> Set-Location HKCU:
PS HKCU:\> Set-ItemProperty .\Software\sample -Value "new value" -name sample
PS HKCU:\> Pop-Location
PS C:\>
```

When you know that a registry property value exists, the solution is simple: Use the *Set-ItemProperty* cmdlet and assign a new value. The following example saves the current working location, changes the new working location to the HSG registry key, uses the *Set-ItemProperty* cmdlet to assign new values, and then uses the *Pop-Location* cmdlet to return to the original working location:

```
PS C:\> Push-Location
PS C:\> Set-Location HKCU:\Software\sample
PS HKCU:\Software\sample> Set-ItemProperty . newproperty "mynewvalue"
PS HKCU:\Software\sample> Pop-Location
PS C:\>
```

NOTE **The preceding code relies on positional parameters for the *Set-ItemProperty* cmdlet. The first parameter is *path*. Because the *Set-Location* cmdlet set the working location to the HSG registry key, a period identifies the path as the current directory. The second parameter is the *Name* of the registry property to change; in this example, it is *newproperty*. The last parameter is *value,* and that defines the value to assign to the registry property. In this example, it is *mynewvalue*. The command, with complete parameter names, is *Set-ItemProperty -Path . -Name newproperty -Value mynewvalue*. The quotation marks in the code are not required, but do not harm anything either.**

Of course, all the pushing, popping, and setting of locations are not really required. It is entirely possible to change the registry property value from any location within the Windows PowerShell provider subsystem.

Use the *Set-ItemProperty* cmdlet to assign a new value. Ensure you specify the complete path to the registry key.

The following example shows how to use the *Set-ItemProperty* cmdlet to change a registry property value without first navigating to the registry drive:

```
PS C:\> Set-ItemProperty -Path HKCU:\Software\test -Name newproperty -Value anewvalue
```

Dealing with a missing registry property value

If you need to set a registry property value, you can set the value of the property easily by using the *Set-ItemProperty* cmdlet. But what if the registry property does not exist? How do you set the property value then? You can still use the *Set-ItemProperty* cmdlet to set a registry property value, even if the registry property does not exist:

```
Set-ItemProperty -Path HKCU:\Software\sample -Name missingproperty -Value avalue
```

To determine if a registry key exists is easy: simply use the *Test-Path* cmdlet. It returns *True* if the key exists and *False* if it does not exist. The following example shows this technique:

```
PS C:\> Test-Path HKCU:\Software\sample
True
PS C:\> Test-Path HKCU:\Software\sample\newproperty
False
```

Unfortunately, this technique does not work for a registry key property. It always returns *False,* even if the registry property exists:

```
PS C:\> Test-Path HKCU:\Software\sample\newproperty
False
PS C:\> Test-Path HKCU:\Software\sample\bogus
False
```

Therefore, if you do not want to overwrite a registry key property if it already exists, you need a way to determine if the registry key property exists. Using the *Test-Path* cmdlet does not work.

Use the If statement and the *Get-ItemProperty* cmdlet to retrieve the value of the registry key property. Specify the erroraction (*ea* is an alias) of silentlycontinue (0 is the enumeration value). In the *scriptblock* for the If statement, display a message that the registry property exists or simply exit. In the Else statement, call the *Set-ItemProperty* to create and set the value of the registry key property. The following example shows this technique:

```
if((Get-ItemProperty HKCU:\Software\sample -Name bogus -ea 0).bogus) {'Propertyalready
exists'}
ELSE { Set-ItemProperty -Path HKCU:\Software\sample -Name bogus -Value 'initial value'}
```

Understanding the Variable provider

The Variable provider provides access to the variables that are defined within Windows PowerShell. These variables include both user-defined variables, such as $mred, and system-defined variables, such as $host. You can obtain a listing of the cmdlets designed to work specifically with variables by using the *Get-Help* cmdlet and specifying the asterisk (*) variable. To return only cmdlets, you can use the *Where-Object* cmdlet and filter on the category that is equal to cmdlet. The following example shows this command:

```
Get-Help *variable | Where-Object category -eq "cmdlet"
```

You can also use the *Where-Object* command, as shown in the following example:

```
Get-Help -Name Variable -Category cmdlet
```

The resulting list contains five cmdlets but is a little jumbled and difficult to read. So let's modify the preceding command and specify the properties to return. To do this, use the Up arrow key and pipeline the returned object into the *Format-List* cmdlet. Add the three properties we are interested in: *Name*, *Category*, and *Synopsis*. The following example shows the revised command:

```
Get-Help *variable | Where-Object {$_.category -eq "cmdlet"} |
Format-List name, category, synopsis
```

The resulting output is much easier to read and understand:

```
Name     : Get-Variable
Category : Cmdlet
Synopsis : Gets the variables in the current console.

Name     : New-Variable
Category : Cmdlet
Synopsis : Creates a new variable.

Name     : Set-Variable
Category : Cmdlet
Synopsis : Sets the value of a variable. Creates the variable if one with the requested
name does not exist.

Name     : Remove-Variable
Category : Cmdlet
Synopsis : Deletes a variable and its value.
```

```
Name     : Clear-Variable
Category : Cmdlet
Synopsis : Deletes the value of a variable.
```

Summary

This chapter discussed how to use the various Windows PowerShell providers that ship with Windows PowerShell 3.0. Specifically, we examined the Alias provider, Certificate provider, Environment provider, File System provider, Function provider, Registry provider, and Variable provider.

CHAPTER 7

Using Windows PowerShell remoting

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

When you need to use Windows PowerShell on your local computer, it is pretty easy: You open the Windows PowerShell console or the Windows PowerShell ISE, and you run a command or a series of commands. Assuming you have rights to make the changes in the first place, it just works. But what if the change you need to make must be enacted on a hundred or a thousand computers? In the past, these types of changes required expensive specialized software packages, but with Windows PowerShell 3.0 running a command on a remote computer is as easy as running the command on your local computer; in some cases, it is even easier.

Using Windows PowerShell remoting

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

Classic remoting

Classic remoting relies on protocols such as the Distributed Component Object Model (DCOM) and remote procedure call (RPC) to make connections to remote machines. Traditionally, these techniques require opening many ports in the firewall and starting various services the different cmdlets utilize. To find the Windows PowerShell cmdlets that natively support remoting, use the *Get-Help* cmdlet. Specify a value of computername for the parameter of the *Get-Help* cmdlet. This command produces a nice list of all cmdlets that have native support for remoting. The following example shows the command and associated

output (this command does not display all cmdlets with support for computername unless the associated modules are preloaded):

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

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

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

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

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

TimeCreated           ProviderName                          Id Message
-----------           ------------                          -- -------
7/1/2012 11:54:14 AM MSExchange ADAccess                 2080 Process MAD.EXE (...
```

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

```
PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential
nwtraders\administrator
Get-WinEvent : The RPC server is unavailable
At linE:1 chaR:1
+ Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential iam
...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecifieD: (:) [Get-WinEvent], EventLogException
    + FullyQualifiedErrorId : System.Diagnostics.Eventing.Reader.EventLogException,
   Microsoft.PowerShell.Commands.GetWinEventCommand
```

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

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

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


PS C:\>
```

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

Configuring Windows PowerShell remoting

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

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

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

```
Enable-PSRemoting -Force
```

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

```
PS C:\> Enable-PSRemoting

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

Do you want to continue?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y
WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.

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


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

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

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name:
microsoft.powershell32 SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will
allow selected users to remotely run Windows PowerShell commands on this computer".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y
PS C:\>
```

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

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


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

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

Running commands

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

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


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



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

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

The good thing is that when using the Windows PowerShell transcript tool through *Start-Transcript*, the transcript tool captures output from the remote Windows PowerShell session as well as output from the local session. Indeed, all commands typed appear in the transcript.

The following commands illustrate beginning a transcript, entering a remote Windows PowerShell session, typing a command, exiting the session, and stopping the transcript:

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


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


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

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

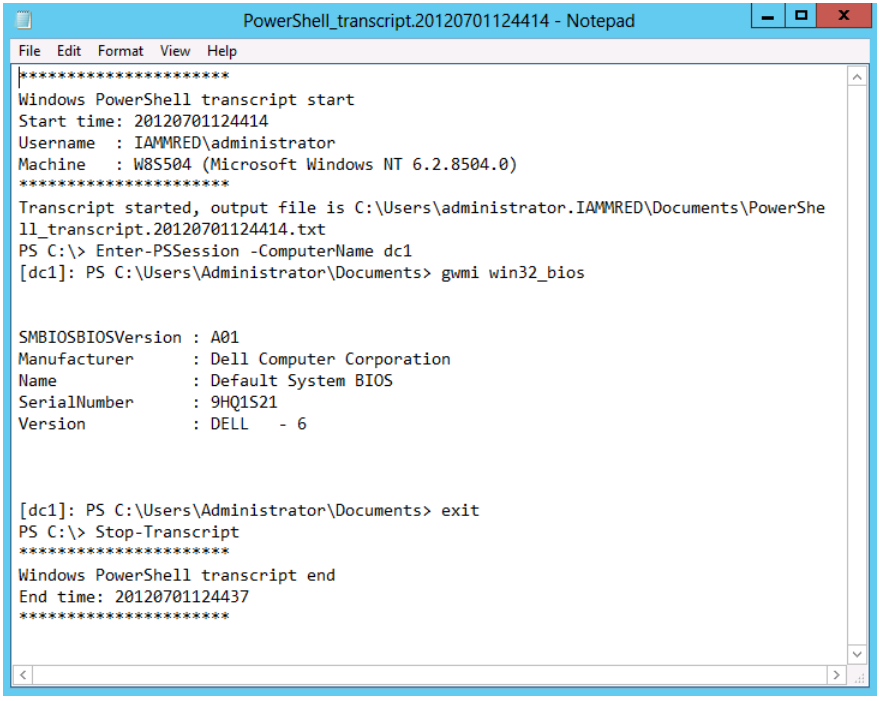


FIGURE 7-1 The transcript tool works in remote Windows PowerShell sessions as well as in local Windows PowerShell console sessions.

Running a single Windows PowerShell command

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

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

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

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

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

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

Running a single command against multiple computers

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

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

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

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

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

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

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

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

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

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

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


PS C:\>
```

Creating a persisted connection

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

```
PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential iammred\administrator
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> hostname
dc1
[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> gps | select -Last 1

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    292       9    39536      50412   158     1.97   2332 wsmprovhost


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

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


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

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

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

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


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

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

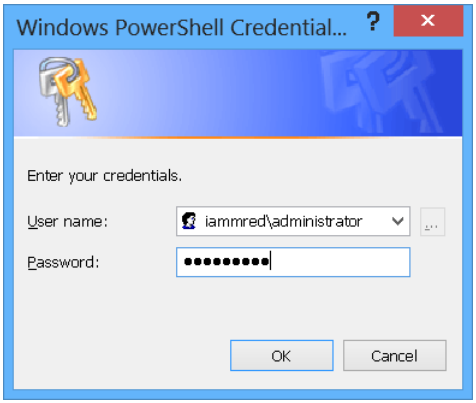


FIGURE 7-2 Store remote credentials in a variable populated through the *Get-Credential* cmdlet.

Once you have stored the credentials in a variable, create another variable to store the remote computer names. Next, use the *New-PSSession* cmdlet to create a new Windows PowerShell session using the computer names stored in the computer name variable and the credentials stored in the credential variable. To be able to reuse the Windows PowerShell remote session, store the newly created Windows PowerShell session in a variable as well. The following example illustrates storing the credentials, computer names, and newly created Windows PowerShell session:

```
$cred = Get-Credential -Credential iammred\administrator
$cn = "ex1","dc3"
$ps = New-PSSession -ComputerName $cn -Credential $cred
```

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

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

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

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

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

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

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

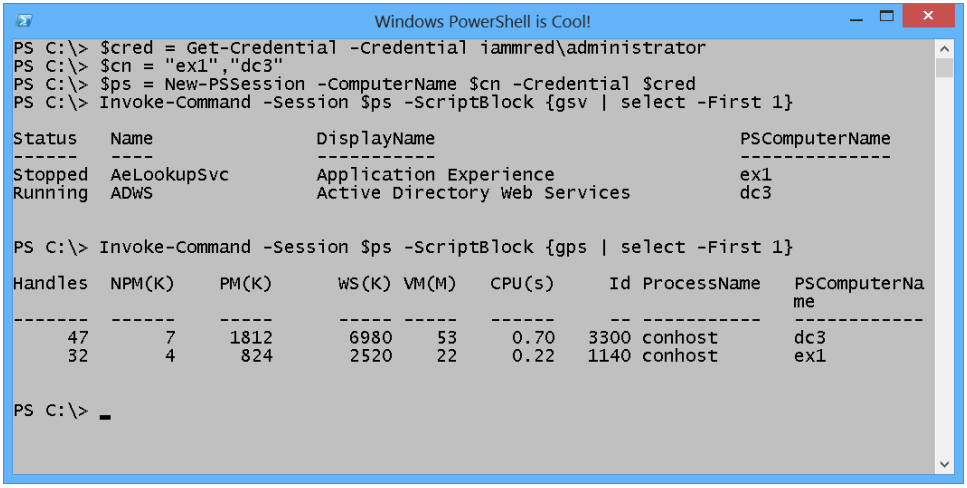


FIGURE 7-3 By creating and by storing a remote Windows PowerShell connection, it becomes easy to run commands against multiple computers.

Troubleshooting Windows PowerShell remoting

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

```
PS C:\> Test-WSMan


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

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

```
PS C:\> Test-WSMan -ComputerName dc3


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

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

```
PS C:\> Test-WSMan -ComputerName dc1


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

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

```
PS C:\> $ps | Get-PSSession

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

To focus on a specific session, reference the session by either ID or by Name. Send the returned session object over the pipeline to the *Format-List* cmdlet and select all the properties. The following example shows this technique (using *fl* as an alias for the *Format-List* cmdlet):

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


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

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

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

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

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

Summary

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

CHAPTER 8

Using WMI

- Understanding the WMI model
- Working with objects and namespaces
- Querying WMI: The basics

The inclusion of Windows Management Instrumentation (WMI) in virtually every operating system released by Microsoft since Windows NT 4.0 should give you an idea of the importance of this underlying technology. From a network management perspective, many useful tasks can be accomplished using just Windows PowerShell, but to truly begin to unleash the power of scripting, you need to bring in additional tools. This is where WMI comes into play. WMI provides access to many powerful ways of managing Microsoft Windows systems. In this section, we dive into the pieces that make up WMI. We look at several concepts, such as namespaces, providers, and classes, and show how these concepts can aid us in leveraging WMI in our Windows PowerShell scripts.

Understanding the WMI Model

WMI is a hierarchical namespace, in which the layers build on one another like the Lightweight Directory Access Protocol (LDAP) directory used in Active Directory or the file system structure on your hard disk drive. Although it is true that WMI is a hierarchical namespace, the term by itself does not really convey the richness of WMI. The WMI model has three sections: resources, infrastructure, and consumers. These sections have the following uses:

- **WMI resources** Resources include anything that can be accessed by using WMI: the file system, networked components, event logs, files, folders, disks, Active Directory, and so on.
- **WMI infrastructure** The infrastructure comprises three parts: the WMI service, the WMI repository, and the WMI providers. Of these parts, WMI providers are the most important because they provide the means for WMI to gather needed information.
- **WMI consumers** A consumer consumes the data from WMI. A consumer can be a Windows PowerShell cmdlet, a Visual Basic Script (VBScript), an enterprise management software package, or some other tool or utility that executes WMI queries.

Working with objects and namespaces

Let's go back to the idea of a namespace introduced earlier in this chapter. You can think of a namespace as a way to organize or collect data related to similar items. Visualize an old- filing cabinet. Each drawer can represent a particular namespace. Inside this drawer are hanging folders that collect information related to a subset of what the drawer actually holds. For example, at home in my filing cabinet, I have a drawer reserved for information related to my woodworking tools. Inside this particular drawer are hanging folders for my table saw, my planer, my joiner, my dust collector, and so on. In the folder for the table saw is information about the motor, the blades, and the various accessories I purchased for the saw, such as an over-arm blade guard.

WMI organizes the namespaces in a similar fashion as the filing cabinets. It is possible to extend the filing cabinet analogy to include the three components of WMI with which you will work. The three components are namespaces, providers, and classes. The namespaces are the file cabinets. The providers are drawers in the file cabinet. The folders in the drawers of the file cabinet are the WMI classes.

Namespaces contain objects, and these objects contain properties you can manipulate. Let's use a WMI command to illustrate the organization of the WMI namespaces. In the command that follows the *Get-WmiObject* cmdlet is used to make the connection into WMI. The class argument specifies the name of the class. In this example, the class name is *__Namespace* (the WMI class from which all WMI namespaces derive). Yes, you read that class name correctly; it is the word namespace preceded by two underscore characters. A double underscore is used for all WMI system classes because it makes them easy to find in sorted lists; the double underscore when sorted rises to the top of the list. The namespace argument is root because it specifies the root level (the top namespace) in the WMI namespace hierarchy. The following example shows the *Get-WmiObject* class:

```
Get-WmiObject -class __Namespace -namespace root
```

Listing WMI providers

Understanding the namespace assists the network administrator with judiciously applying WMI scripting to her network duties. However, as mentioned earlier, to access information through WMI you must have access to a WMI provider. After implementing the provider, you can gain access to the information the provider makes available.

> ***NOTE*** **Keep in mind that in nearly every case, installation of providers happens in the background through an operating system configuration or management application installation. For example, addition of new roles and features to server SKUs often installs new WMI providers and their attendant WMI classes.**

WMI bases providers on a template class, or a system class called *__provider*. Armed with this information, we can look for instances of the provider class, and we will have a list of all the providers that reside in our WMI namespace. This is exactly what the *Get-WmiProvider* function does.

The *Get-WmiProvider* function begins by assigning the string *Root\cimv2* to the *$wmiNS* variable. This value will be used with the *Get-WmiObject* cmdlet to specify where the WMI query will take place.

The *Get-WmiObject* cmdlet queries WMI. The class parameter limits the WMI query to the *__provider* class. The namespace argument tells the *Get-WmiObject* cmdlet to look only in the *Root\cimv2* WMI namespace. The array of objects returned from the *Get-WmiObject* cmdlet pipelines to the *Sort-Object* cmdlet, where the listing of objects alphabetize based on the *name* property. After this process is completed, the reorganized objects pipeline to the *Select-Object* cmdlet, where the name of each provider displays. The following example shows this process:

```
Get-WmiObject -class __Provider -namespace root\cimv2 |
  Sort-Object -property Name |
  Select-Object name
```

Working with WMI classes

In addition to working with namespaces, the inquisitive network administrator will want to explore the concept of classes. In WMI parlance, you have core classes, common classes, and dynamic classes. *Core classes* represent managed objects that apply to all areas of management. These classes provide a basic vocabulary for analyzing and describing managed systems. Two examples of core classes are parameters and the *SystemSecurity* class. *Common classes* are extensions to the core classes and represent managed objects that apply to specific management areas. However, common classes are independent from a particular implementation or technology. The *CIM_UnitaryComputerSystem* is an example of a common class. Network administrators do not use core and common classes because they serve as templates from which other classes derive, and as such are mainly of interest to developers of management applications.

Many of the classes stored in *Root\cimv2*, therefore, are abstract classes and are of use as templates used in creating other WMI classes. However, a few classes in *Root\cimv2* are *dynamic classes* used to hold actual information. The important aspect to remember about dynamic classes is that providers generate instances of a dynamic class, and therefore dynamic WMI classes are more likely to retrieve "live" data from the system.

To produce a simple listing of WMI classes, you can use the *Get-WmiObject* cmdlet and specify the *list* argument:

```
Get-WmiObject -list
```

A partial output from the previous command is shown in the following example:

```
Win32_TSGeneralSetting                  Win32_TSPermissionsSetting
Win32_TSClientSetting                   Win32_TSEnvironmentSetting
Win32_TSNetworkAdapterListSetting       Win32_TSLogonSetting
Win32_TSSessionSetting                  Win32_DisplayConfiguration
Win32_COMSetting                        Win32_ClassicCOMClassSetting
Win32_DCOMApplicationSetting            Win32_MSIResource
Win32_ServiceControl                    Win32_Property
```

One of the big problems with WMI is finding the WMI class needed to solve a particular problem. With literally thousands of WMI classes installed on even a basic Windows installation, searching through all the classes is difficult at best. While it is possible to search the Microsoft Developer Network (MSDN), a faster solution is to use Windows PowerShell itself. As we just saw, using the list switched parameter produces a listing of all the WMI classes in a particular namespace. It is possible to combine WMI namespaces to use the feature to produce a listing of every WMI class on a computer, but that would only compound an already complicated situation.

A better solution is to stay focused on a single WMI namespace, and to use wildcard characters to assist in finding appropriate WMI classes. For example, you can use the wildcard pattern "*bios*" to find all WMI classes that contain the letters bios in the class name. The following code accomplishes this task:

```
Get-WmiObject -List "*bios*"
```

In fact, you should not use all the classes for direct query. Nevertheless, some of the classes are useful; some of the classes solve the problem. You might ask, "Which ones should I use?" A simple answer, not completely accurate, but something to get you started, is to use only the WMI classes that begin with win32. You can easily modify the previous *Get-WMIObject* query to return only WMI classes that begin with win32. A regular expression pattern looks at the first position of each WMI class name to determine if the characters win32 appear. The special character ^ tells the match operator to begin looking at the beginning of the string. The following example shows the revised code:

```
Get-WmiObject -List "*bios*" | where name -match '^win32'
```

It is also possible to simplify the preceding code by taking advantage of command aliases and the simplified *Where-Object* syntax. In the code that follows, gwmi is an alias for the *Get-WmiObject* cmdlet. The ? symbol is an alias for the *Where-Object* cmdlet, and the name property from the returned ManagementClass objects is examined from each instance crossing the pipeline to see if the regular expression pattern match appears. The shorter syntax appears here:

```
gwmi -list "*bios*" | ? name -match  '^win32'
```

It is even possible to combine the following into a single command and avoid the pipeline altogether. The following command looks for WMI classes that begin with the letters *win32* and are followed by the letters *bios* in a four-letter pattern:

```
gwmi -List 'win32*bios*'
```

Only a few WMI classes return from the preceding command. It is now time to query each WMI class to determine the WMI classes that might be useful. It is certainly possible to choose a class from the list and to query it directly. Using the gwmi alias for the *Get-WmiObject* cmdlet, it is not very much typing. Here is the command to return BIOS information from the local computer:

```
gwmi win32_bios
```

It is also possible to pipeline the results of the query to find WMI classes to a command to query the WMI classes. The long form of the command (using complete cmdlet names) appears here. Keep in mind this is a single-line command, which appears here on two different lines for readability:

```
Get-WmiObject -List "*bios*" | Where-Object { $_.name -match '^win32'} |
ForEach-Object { Get-WmiObject -Class $_.name }
```

The short form of the command uses the alias *gwmi* for *Get-WmiObject*, *?* for the *Where-Object* cmdlet as well as the simplified *Where-Object* syntax, and *%* for the *ForEach-Object* cmdlet. The following example shows the shortened command:

```
gwmi -list "*bios*" | ? name -match  '^win32' | % {gwmi $_.name}
```

Querying WMI: The basics

In most situations, when you use WMI you are performing some sort of query. Even when you are going to set a particular property, you still need to execute a query to return a dataset that enables you to perform the configuration. A *dataset* includes the data that come back to you as the result of a query; that is, it is a set of data. There are several steps involved in performing a basic WMI query:

1. Connect to WMI by using the *Get-WMIObject* cmdlet.
2. Specify a valid WMI class name to query.
3. Specify a value for the namespace; omit the namespace parameter to use the default *root\cimv2* namespace.
4. Specify a value for the computername parameter; omit the computername parameter to use the default value of localhost.

Windows PowerShell makes it easy to query WMI. In fact, at its most basic level the only thing required is gwmi (alias for the *Get-WmiObject* cmdlet) and the WMI class name. The following example shows this simple syntax, along with the associated output:

```
PS C:\> gwmi win32_bios
SMBIOSBIOSVersion : BAP6710H.86A.0064.2011.0504.1711
Manufacturer      : Intel Corp.
Name              : BIOS Date: 05/04/11 17:11:33 Ver: 04.06.04
SerialNumber      :
Version           : INTEL  - 1072009
```

However, there are more properties available on the *Win32_Bios* WMI class than the five displayed in the preceding output. The reason for the limited output displayed from the command is a custom view of the *Win32_Bios* class defined in the Types.ps1xml file that resides in the Windows PowerShell home directory on your system. The following command uses the *Select-String* cmdlet to search the Types.ps1xml file to see if there is any reference to the WMI class *Win32_Bios*:

```
Select-String -Path $pshome\*.ps1xml -SimpleMatch "Win32_Bios"
```

In Figure 8-1, several *Select-String* commands display results when a special format exists for a particular WMI class. The last query, for the *Win32_CurrentTime* WMI class, does not return any results, indicating that no special formatting exists for this class.

FIGURE 8-1 Use the *Select-String* cmdlet to find format XML files for specific WMI classes.

The *Select-String* queries shown in Figure 8-1 indicate there is a special formatting for the *Win32_Bios*, *Win32_DesktopMonitor*, and *Win32_Service* WMI classes. The Types.ps1xml file contains information to Windows PowerShell that tells it how to display a particular WMI class. When an instance of the *Win32_Bios* WMI class appears, Windows PowerShell uses

the DefaultDisplayPropertySet configuration to display only five properties. The following example shows the portion of the Types.ps1xml file that details these five properties:

```
<PropertySet>
    <Name>DefaultDisplayPropertySet</Name>
    <ReferencedProperties>
        <Name>SMBIOSBIOSVersion</Name>
        <Name>Manufacturer</Name>
        <Name>Name</Name>
        <Name>SerialNumber</Name>
        <Name>Version</Name>
    </ReferencedProperties>
</PropertySet>
```

Figure 8-2 shows the complete type definition for the *Win32_Bios* WMI class.

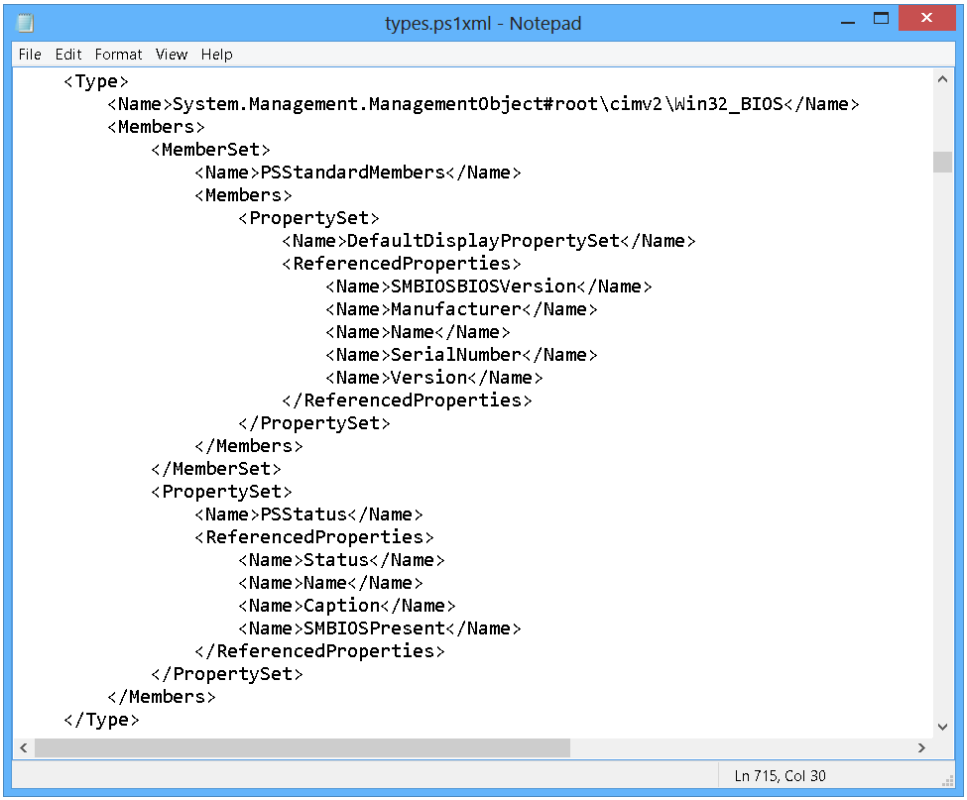


FIGURE 8-2 The Types.PS1XML file contains information that tells Windows PowerShell which properties to display by default.

Special formatting instructions for the *Win32_Bios* WMI class indicate there is an alternative property set available in addition to the DefaultDisplayPropertySet. This additional

property set, named PSStatus, contains four properties. The following example shows the four properties that appear in the PropertySet description:

```
<PropertySet>
    <Name>PSStatus</Name>
    <ReferencedProperties>
        <Name>Status</Name>
        <Name>Name</Name>
        <Name>Caption</Name>
        <Name>SMBIOSPresent</Name>
    </ReferencedProperties>
</PropertySet>
```

Finding the PSStatus property set is more than a simple academic exercise because it can be used directly with Windows PowerShell cmdlets such as *Select-Object* (*select* is an alias), *Format-List* (*fl* is an alias), or *Format-Table* (*ft* is an alias). The following commands illustrate this technique:

```
gwmi win32_bios | select psstatus
gwmi win32_bios | fl psstatus
gwmi win32_bios | ft psstatus
```

Unfortunately, you cannot use the alternate property set PSStatus to select the properties through the property parameter. Therefore, the command that appears here fails:

```
gwmi win32_bios -Property psstatus
```

Tell me everything about everything

When novices first write WMI scripts, they nearly all begin by asking for every property from every instance of a class. For example, the queries will say, "Tell me everything about every process." This is also referred to as the infamous "select * query." This approach often can return an overwhelming amount of data, particularly when you are querying a class such as installed software or processes and threads. Rarely does one need to have so much data. Typically, when looking for installed software, you're looking for information about a particular software package.

There are, however, several occasions when you want to use the "tell me everything about all instances of a particular class" query, including the following:

- During development of a script to see representative data.
- When troubleshooting a more directed query; for example, when you try to filter on a field that does not exist.
- When the returned data are so few that being more precise doesn't make sense.

In the next command, you make a connection to the default namespace in WMI and return all the information about all the shares on a local machine. This is actually good practice because in the past numerous worms have propagated through unsecured shares, and you

might have unused shares around. A user might create a share for a friend and then forget to delete it. The following example shows this connection to the default namespace:

```
Get-WmiObject -Class win32_Share -Property * | Format-List *
```

The data returned contain a number of system properties and other objects not directly related to a share, as shown in the following example:

```
PS C:\> Get-WmiObject -Class win32_Share -Property * | Format-List *


PSComputerName   : EDLT
Status           : OK
Type             : 2147483648
Name             : ADMIN$
__GENUS          : 2
__CLASS          : Win32_Share
__SUPERCLASS     : CIM_LogicalElement
__DYNASTY        : CIM_ManagedSystemElement
__RELPATH        : Win32_Share.Name="ADMIN$"
__PROPERTY_COUNT : 10
__DERIVATION     : {CIM_LogicalElement, CIM_ManagedSystemElement}
__SERVER         : EDLT
__NAMESPACE      : root\cimv2
__PATH           : \\EDLT\root\cimv2:Win32_Share.Name="ADMIN$"
AccessMask       :
AllowMaximum     : True
Caption          : Remote Admin
Description      : Remote Admin
InstallDate      :
MaximumAllowed   :
Path             : C:\WINDOWS
Scope            : System.Management.ManagementScope
Options          : System.Management.ObjectGetOptions
ClassPath        : \\EDLT\root\cimv2:Win32_Share
Properties       : {AccessMask, AllowMaximum, Caption, Description...}
SystemProperties : {__GENUS, __CLASS, __SUPERCLASS, __DYNASTY...}
Qualifiers       : {dynamic, Locale, provider, UUID}
Site             :
Container        :
<… Output truncated …>
```

The default output for this class is probably more immediately useful. The following example shows the output:

```
PS C:\> Get-WmiObject -Class win32_Share

Name                        Path                        Description
----                        ----                        -----------
ADMIN$                      C:\WINDOWS                  Remote Admin
C$                          C:\                         Default share
IPC$                                                    Remote IPC
Users                       C:\Users
```

Rather than specifying the -class and -property parameters to obtain all the WMI information related to the *Win32_Share* WMI class, it is possible to use a SQL type of syntax. This is the same type of syntax used in VBScript and in other scripting languages. The advantage is that it might be more readable, and there are lots of examples of the syntax available on scripting repositories (such as the TechNet Script Center Script Repository). The following example illustrates creating the query and using it in a command:

```
PS C:\> $query = "Select * from Win32_Share"
PS C:\> Get-WmiObject -Query $query

Name                          Path                          Description
----                          ----                          -----------
ADMIN$                        C:\WINDOWS                    Remote Admin
C$                            C:\                           Default share
IPC$                                                        Remote IPC
Users                         C:\Users
```

Of course, it is possible to perform this query in a single line instead of storing the WMI query in a variable. The following example illustrates this technique:

```
Get-WmiObject -Query "Select * from Win32_Share"
```

Tell me selected things about everything

The next level of sophistication from using Select * is to return only the properties you are interested in. This is a more efficient strategy. For instance, in the preceding example you entered Select * and were returned a lot of data you weren't necessarily interested in. Suppose you want to know only what shares are on each machine. You first make a connection to WMI by using the Get-WmiObject cmdlet. Next you use the query argument to supply the WMI query to the Get-WmiObject cmdlet. In the query, use the Select statement to choose the specific property you are interested in; for example, Select name. In the query, use the From statement to indicate the class from which you want to retrieve data—for example, From Win32_Share.

Only two small changes from the previous query are required to enable garnering specific data through the WMI script. In place of the asterisk in the Select statement assigned at the beginning of the script, substitute the property you want. In this case, only the names of the shares are required. The following example shows the first revision:

```
PS C:\> Get-WmiObject -Query "Select name from win32_share"


__GENUS          : 2
__CLASS          : Win32_Share
__SUPERCLASS     :
__DYNASTY        :
__RELPATH        : Win32_Share.Name="ADMIN$"
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
```

```
__NAMESPACE      :
__PATH           :
Name             : ADMIN$
PSComputerName   :
<…Output Truncated …>
```

The second change is to eliminate all unwanted system properties from the Output section. The strange thing here is the way that Windows PowerShell works. In the Select statement, we selected only the *name* property. However, if we were to print out the results without further refinement, we would retrieve unwanted system properties as well. By using the Format-List cmdlet and selecting only the property name, you eliminate the unwanted excess. The following example shows this technique:

```
PS C:\> Get-WmiObject -Query "Select name from win32_share" | Format-List name


name : ADMIN$

name : C$

name : IPC$

name : Users
```

This same technique (of selecting the name from each share) appears in the following code using the -Property parameter:

```
PS C:\> gwmi win32_share -Property name | fl name


name : ADMIN$

name : C$

name : IPC$

name : Users
```

Tell me everything about some things

In many situations, you will want to limit the data you return to a specific instance of that class in the dataset. If you go back to your query and add a Where clause to the Select statement, you'll be able to greatly reduce the amount of information returned by the query. Notice that in the value associated with the wmiQuery, you added a dependency that indicated you wanted only information with the share name C$. This value is not case sensitive, but it must be surrounded with single quotation marks, as you can see in the wmiQuery string in the following script. These single quotation marks are important because they tell WMI that the value is a string value and not some other programmatic item. The code illustrates returning

all available properties from a specific share, the admin share of the C drive. There are two ways to do this. The first way uses the Where clause in the WMI query:

```
PS C:\> $wmiQuery = "Select * from win32_share where name='c$'"
PS C:\> Get-WmiObject -Query $wmiQuery | Format-List *


PSComputerName   : EDLT
Status           : OK
Type             : 2147483648
Name             : C$
__GENUS          : 2
__CLASS          : Win32_Share
__SUPERCLASS     : CIM_LogicalElement
__DYNASTY        : CIM_ManagedSystemElement
__RELPATH        : Win32_Share.Name="C$"
__PROPERTY_COUNT : 10
__DERIVATION     : {CIM_LogicalElement, CIM_ManagedSystemElement}
__SERVER         : EDLT
__NAMESPACE      : root\cimv2
__PATH           : \\EDLT\root\cimv2:Win32_Share.Name="C$"
AccessMask       :
AllowMaximum     : True
Caption          : Default share
Description      : Default share
InstallDate      :
MaximumAllowed   :
Path             : C:\
Scope            : System.Management.ManagementScope
Options          : System.Management.ObjectGetOptions
ClassPath        : \\EDLT\root\cimv2:Win32_Share
Properties       : {AccessMask, AllowMaximum, Caption, Description...}
SystemProperties : {__GENUS, __CLASS, __SUPERCLASS, __DYNASTY...}
Qualifiers       : {dynamic, Locale, provider, UUID}
Site             :
Container        :
```

The second way to reduce the number of instances returned by a WMI query is to use the -Filter parameter. This works in a manner similar to the Where clause (except that it does not use the keyword Where):

```
Get-WmiObject -Class win32_share -Filter "name = 'c$'" | Format-List *
```

Tell me selected things about some things

The most specific WMI query is one that returns only a few properties from a few instances of the class. For example, you might be interested in the directory provided by a specific share. To do this, you need to specify which properties you wish to retrieve as well as specify the name of a particular share of interest. This involves both selecting the properties to return as well as limiting the instances returned. Use the following WMI query:

```
PS C:\> $query = "Select name, path from win32_share where name = 'users'"
PS C:\> Get-WmiObject -Query $query | format-table name, path

name                                    path
----                                    ----
Users                                   C:\Users
```

You can also omit the -Query parameter and specify the -Filter and the -Property parameters. This following example shows this revision:

```
Get-WmiObject -Class win32_share -Property name, path -Filter "name = 'users" |
format-table name, path
```

Summary

This chapter began with a basic overview of WMI. In the overview, you learned about WMI classes, providers, and namespaces. The chapter concluded by querying WMI. We examined two different ways of querying WMI: by using an SQL type of syntax and by using specific parameters from the *Get-WMIObject* cmdlet. In addition to two different ways of querying WMI, we also examined four ways of returning WMI data.

CHAPTER 9

Using CIM

- Using CIM cmdlets to explore WMI classes
- Retrieving WMI instances
- Working with associations

One of the new and exciting changes to WMI in Windows PowerShell 3.0 is the introduction of the Common Information Model (CIM). The CIM cmdlets provide a new way to retrieve existing WMI. The advantages are numerous. First, you will probably notice that the CIM cmdlets return WMI information faster than does the *Get-WMiObject* cmdlet. Second, because the CIM cmdlets use WinRM (instead of the legacy DCOM) for a network transport, it means you can immediately use CIM to query your remote Windows Server 2012 computer. It also means you can easily use CIM for remote management because WinRM is easily configured. See Chapter 7, "Using Windows PowerShell Remoting" for more information about WinRM. Lastly, the CIM cmdlets provide several cmdlets that promote WMI discovery. All in all, CIM is a good thing.

Using CIM cmdlets to explore WMI classes

CIM cmdlets support multiple ways of exploring WMI. They work well when working in an interactive fashion. For example, tab expansion expands the namespace when you use CIM cmdlets, thereby permitting you to explore namespaces that might not otherwise be very discoverable. You can even drill down into namespaces by using this technique. All CIM classes support tab expansion of the namespace parameter, but to explore WMI classes you should use the *Get-CimClass* cmdlet.

NOTE **The default WMI namespace on all operating systems after Windows NT 4.0 is *Root/Cimv2*. Therefore, all the CIM cmdlets default to *Root/Cimv2*. The only time you need to change the default WMI namespace (through the namespace parameter) is when you need to use a WMI class from a non-default WMI namespace.**

Using the classname parameter

Using the *Get-CimClass* cmdlet, you can use wildcards for the classname parameter to enable you to quickly identify potential WMI classes for perusal. You can also use wildcards for the qualifiername parameter. In the following example, the *Get-CimClass* cmdlet looks for WMI classes related to computers:

```
PS C:\> Get-CimClass -ClassName *computer*


   NameSpace: ROOT/CIMV2

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
Win32_ComputerSystemEvent           {}                   {SECURITY_DESCRIPTOR, TIME_
CR...
Win32_ComputerShutdownEvent         {}                   {SECURITY_DESCRIPTOR, TIME_
CR...
CIM_ComputerSystem                  {}                   {Caption, Description,
Instal...
CIM_UnitaryComputerSystem           {SetPowerState}      {Caption, Description,
Instal...
Win32_ComputerSystem                {SetPowerState, R... {Caption, Description,
Instal...
CIM_ComputerSystemResource          {}                   {GroupComponent, PartComponent}
CIM_ComputerSystemMappedIO          {}                   {GroupComponent, PartComponent}
CIM_ComputerSystemDMA               {}                   {GroupComponent, PartComponent}
CIM_ComputerSystemIRQ               {}                   {GroupComponent, PartComponent}
Win32_ComputerSystemProcessor       {}                   {GroupComponent, PartComponent}
CIM_ComputerSystemPackage           {}                   {Antecedent, Dependent}
Win32_ComputerSystemProduct         {}                   {Caption, Description,
Identi...
Win32_NTLogEventComputer            {}                   {Computer, Record}
```

> ***NOTE*** **If you try to use a wildcard for the *classname* parameter of the *Get-CimInstance* cmdlet, an error returns because the parameter design does not permit wildcard characters.**

Finding WMI class methods

If you want to find WMI classes related to processes that contain a method that begins with the letters *term**, you use a command similar to the one in the following example:

```
PS C:\> Get-CimClass -ClassName *process* -MethodName term*


   NameSpace: ROOT/cimv2
```

```
CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
Win32_Process                       {Create, Terminat... {Caption, Description,
Instal...
```

To find all WMI classes related to processes that expose any methods, use the command in the following example:

```
PS C:\> Get-CimClass -ClassName *process* -MethodName *


   NameSpace: ROOT/cimv2

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
Win32_Process                       {Create, Terminat... {Caption, Description,
Instal...
CIM_Processor                       {SetPowerState, R... {Caption, Description,
Instal...
Win32_Processor                     {SetPowerState, R... {Caption, Description,
Instal...
```

To find any WMI class in the *root/cimv2* WMI namespace that exposes a method called create, use the command in the following example:

```
PS C:\> Get-CimClass -ClassName * -MethodName create


   NameSpace: ROOT/cimv2

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
Win32_Process                       {Create, Terminat... {Caption, Description,
Instal...
Win32_ScheduledJob                  {Create, Delete}     {Caption, Description,
Instal...
Win32_DfsNode                       {Create}             {Caption, Description,
Instal...
Win32_BaseService                   {StartService, St... {Caption, Description,
Instal...
Win32_SystemDriver                  {StartService, St... {Caption, Description,
Instal...
Win32_Service                       {StartService, St... {Caption, Description,
Instal...
Win32_TerminalService               {StartService, St... {Caption, Description,
Instal...
Win32_Share                         {Create, SetShare... {Caption, Description,
Instal...
Win32_ClusterShare                  {Create, SetShare... {Caption, Description,
Instal...
Win32_ShadowCopy                    {Create, Revert}     {Caption, Description,
Instal...
Win32_ShadowStorage                 {Create}             {AllocatedSpace, DiffVolume,
...
```

Filtering classes by qualifier

To find WMI classes that possess a particular qualifier, use the qualifier parameter. For example, the following command finds WMI classes that relate to computers and have the supportsupdate WMI qualifier:

```
PS C:\> Get-CimClass -ClassName *computer* -QualifierName *update


   NameSpace: ROOT/cimv2

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
Win32_ComputerSystem                {SetPowerState, R... {Caption, Description,
Instal...
```

The parameters can be combined to produce powerful searches that without using the CIM cmdlets would require rather complicated scripting. For example, the following command finds all WMI classes in the *root/Cimv2* namespace that have the singleton qualifier and also expose a method:

```
PS C:\> Get-CimClass -ClassName * -QualifierName singleton -MethodName *


   NameSpace: ROOT/cimv2

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
__SystemSecurity                    {GetSD, GetSecuri... {}
Win32_OperatingSystem               {Reboot, Shutdown... {Caption, Description,
Instal...
Win32_OfflineFilesCache             {Enable, RenameIt... {Active, Enabled, Location}
```

One qualifier that is important to review is the deprecated qualifier. Deprecated WMI classes are not recommended for use because they are being phased out. Using the *Get-CimClass* cmdlet, it is easy to spot these WMI classes. The following example shows this technique:

```
PS C:\> Get-CimClass * -QualifierName deprecated


   NameSpace: ROOT/cimv2

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
Win32_PageFile                      {TakeOwnerShip, C... {Caption, Description,
Instal...
Win32_DisplayConfiguration          {}                   {Caption, Description,
Settin...
Win32_DisplayControllerConfigura... {}                   {Caption, Description,
Settin...
Win32_VideoConfiguration            {}                   {Caption, Description,
Settin...
Win32_AllocatedResource             {}                   {Antecedent, Dependent}
```

Using this technique, it is easy to find association classes. More information about working with WMI association classes appears in the "Working with associations" section later in this chapter.

The following code finds all the WMI classes in the *root/cimv2* WMI namespace that relate to sessions. In addition, it looks for the association qualifier. Luckily, you can use wildcards for the qualifier names, and therefore the following code uses assoc* instead of association:

```
PS C:\> Get-CimClass -ClassName *session* -QualifierName assoc*


   NameSpace: ROOT/cimv2

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
Win32_SubSession                    {}                   {Antecedent, Dependent}
Win32_SessionConnection             {}                   {Antecedent, Dependent}
Win32_LogonSessionMappedDisk        {}                   {Antecedent, Dependent}
Win32_SessionResource               {}                   {Antecedent, Dependent}
Win32_SessionProcess                {}                   {Antecedent, Dependent}
```

One qualifier you should definitely look for is the dynamic qualifier. This is because it is unsupported to query abstract WMI classes. Therefore, when looking for WMI classes you will want to ensure that at some point you run your list through the dynamic filter. In the following example, three WMI classes return that are related to time:

```
PS C:\> Get-CimClass -ClassName *time


   NameSpace: ROOT/cimv2

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
Win32_CurrentTime                   {}                   {Day, DayOfWeek, Hour,
Millis...
Win32_LocalTime                     {}                   {Day, DayOfWeek, Hour,
Millis...
Win32_UTCTime                       {}                   {Day, DayOfWeek, Hour,
Millis...
```

By adding the query for the qualifier, the appropriate WMI classes are identified. One class is abstract and the other two are dynamic classes that could prove to be useful. This following example shows where first, the dynamic qualifier is used and second, where the abstract qualifier appears:

```
PS C:\> Get-CimClass -ClassName *time -QualifierName dynamic


   NameSpace: ROOT/cimv2
```

```
CimClassName                    CimClassMethods      CimClassProperties
------------                    ---------------      ------------------
Win32_LocalTime                 {}                   {Day, DayOfWeek, Hour,
Millis...
Win32_UTCTime                   {}                   {Day, DayOfWeek, Hour,
Millis...


PS C:\> Get-CimClass -ClassName *time -QualifierName abstract


   NameSpace: ROOT/cimv2

CimClassName                    CimClassMethods      CimClassProperties
------------                    ---------------      ------------------
Win32_CurrentTime               {}                   {Day, DayOfWeek, Hour,
Millis...
```

Retrieving WMI instances

To query for WMI data, use the *Get-CimInstance* cmdlet. The easiest way to use the *Get-CimInstance* cmdlet is to query for all properties and all instances of a particular WMI class on the local machine. This is extremely easy to do. The following command illustrates returning BIOS information from the local computer:

```
PS C:\> Get-CimInstance win32_bios


SMBIOSBIOSVersion : 090004
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32  Ver: 09.00.04
SerialNumber      : 4429-0046-2083-1237-7579-8937-43
Version           : VRTUAL - 3000919
```

The *Get-CimInstance* cmdlet returns the entire WMI object, but it honors the format*.xml files that Windows PowerShell uses to determine which properties are displayed by default for a particular WMI class. The following command shows the properties available from the *Win32_Bios* WMI class:

```
PS C:\> $b = Get-CimInstance win32_bios
PS C:\> $b.CimClass.CimClassProperties | fw name -Column 3


Caption                     Description                 InstallDate
Name                        Status                      BuildNumber
CodeSet                     IdentificationCode          LanguageEdition
Manufacturer                OtherTargetOS               SerialNumber
SoftwareElementID           SoftwareElementState        TargetOperatingSystem
Version                     PrimaryBIOS                 BiosCharacteristics
BIOSVersion                 CurrentLanguage             InstallableLanguages
ListOfLanguages             ReleaseDate                 SMBIOSBIOSVersion
SMBIOSMajorVersion          SMBIOSMinorVersion          SMBIOSPresent
```

Reducing returned properties and instances

To limit the amount of data returned from a remote connection, reduce the number of properties returned as well as the number of instances. To reduce properties, use the property parameter. To reduce the number of returned instances use the filter parameter. The following command uses gcim, which is an alias for the *Get-CimInstance* cmdlet. It also abbreviates the classname parameter and the filter parameter. As seen here, the command returns only the name and state from the bits service. The default output, however, shows all the property names as well as the system properties. As shown here, however, only the two selected properties contain data:

```
PS C:\> gcim -clas win32_service -Property name, state -Fil "name = 'bits'"



Name                    : BITS
Status                  :
ExitCode                :
DesktopInteract         :
ErrorControl            :
PathName                :
ServiceType             :
StartMode               :
Caption                 :
Description             :
InstallDate             :
CreationClassName       :
Started                 :
SystemCreationClassName :
SystemName              :
AcceptPause             :
AcceptStop              :
DisplayName             :
ServiceSpecificExitCode :
StartName               :
State                   : Running
TagId                   :
CheckPoint              :
ProcessId               :
WaitHint                :
PSComputerName          :
CimClass                : root/cimv2:Win32_Service
CimInstanceProperties   : {Caption, Description, InstallDate, Name...}
CimSystemProperties     : Microsoft.Management.Infrastructure.CimSystemProperties
```

Cleaning up output from the command

To produce a cleaner output, send the selected data to the *Format-Table* cmdlet. This is easy to do because ft is an alias for the *Format-Table* cmdlet:

```
PS C:\> gcim -clas win32_service -Property name, state -Fil "name = 'bits'" | ft name,
state

name                                    state
----                                    -----
BITS                                    Running
```

Make sure you choose properties you have already selected in the property parameter or else they will not display. In the command appearing here, the status property is selected in the *Format-Table* cmdlet. There is a status property on the *Win32_Service* WMI class, but it was not chosen when the properties were selected:

```
PS C:\> gcim -clas win32_service -Property name, state -Fil "name = 'bits'" | ft name,
state, status

name                    state                   status
----                    -----                   ------
BITS                    Running
```

The *Get-CimInstance* cmdlet does not accept a wildcard parameter for property names (neither does the *Get-WmiObject* cmdlet for that matter). One thing that can simplify some of your coding is to put your property selection into a variable. This permits you to use the same property names in both the *Get-CimInstance* cmdlet and in your *Format-Table* (or *Format-List,* or *Select-Object,* or whatever you are doing after you get your WMI data) without having to type things twice. The following example shows this technique:

```
PS C:\> $property = "name","state","startmode","startname"
PS C:\> gcim -clas win32_service -Pro $property -fil "name = 'bits'" | ft $property -A

name state   startmode startname
---- -----   --------- ---------
BITS Running Manual    LocalSystem
```

Working with associations

In the old-fashioned VBScript days, working with association classes was extremely complicated. This is unfortunate because WMI association classes are extremely powerful and useful. Earlier versions of Windows PowerShell simplified working with association classes, primarily because it simplified working with WMI data in general. However, figuring out how to utilize the Windows PowerShell advantage was still pretty much an advanced technique. Luckily, in Windows PowerShell 3.0 we have the CIM classes that introduce the *Get-CimAssociatedInstance* cmdlet.

The first thing to do is to retrieve a CIM instance and store it in a variable. In the following example, instances of the *Win32_LogonSession* WMI class are retrieved and stored in the *$logon* variable. Next, the *Get-CimAssociatedInstance* cmdlet is used to retrieve instances associated with this class. To see what type of objects will return from the command the results pipe to the *Get-Member* cmdlet. Two WMI classes return: the *Win32_UserAccount* and all processes that are related to that user account in the form of instances of the *Win32_Process* cmdlet:

```
PS C:\> $logon = Get-CimInstance win32_logonsession
PS C:\> Get-CimAssociatedInstance $logon | Get-Member


   TypeName: Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_
UserAccount

Name                       MemberType  Definition
----                       ----------  ----------
Clone                      Method      System.Object ICloneable.Clone()
Dispose                    Method      void Dispose(), void IDisposable.Dispose()
Equals                     Method      bool Equals(System.Object obj)
GetCimSessionComputerName  Method      string GetCimSessionComputerName()
GetCimSessionInstanceId    Method      guid GetCimSessionInstanceId()
GetHashCode                Method      int GetHashCode()
GetObjectData              Method      void GetObjectData(System.Runtime.
Serialization....
GetType                    Method      type GetType()
ToString                   Method      string ToString()
AccountType                Property    uint32 AccountType {get;}
Caption                    Property    string Caption {get;}
Description                Property    string Description {get;}
Disabled                   Property    bool Disabled {get;set;}
Domain                     Property    string Domain {get;}
FullName                   Property    string FullName {get;set;}
InstallDate                Property    CimInstance#DateTime InstallDate {get;}
LocalAccount               Property    bool LocalAccount {get;set;}
Lockout                    Property    bool Lockout {get;set;}
Name                       Property    string Name {get;}
PasswordChangeable         Property    bool PasswordChangeable {get;set;}
PasswordExpires            Property    bool PasswordExpires {get;set;}
PasswordRequired           Property    bool PasswordRequired {get;set;}
PSComputerName             Property    string PSComputerName {get;}
SID                        Property    string SID {get;}
SIDType                    Property    byte SIDType {get;}
Status                     Property    string Status {get;}
PSStatus                   PropertySet PSStatus {Status, Caption, PasswordExpires}


   TypeName: Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_Process

Name                       MemberType     Definition
----                       ----------     ----------
Handles                    AliasProperty  Handles = Handlecount
ProcessName                AliasProperty  ProcessName = Name
VM                         AliasProperty  VM = VirtualSize
WS                         AliasProperty  WS = WorkingSetSize
```

```
Clone                       Method      System.Object ICloneable.Clone()
Dispose                     Method      void Dispose(), void IDisposable.Dispose()
Equals                      Method      bool Equals(System.Object obj)
GetCimSessionComputerName   Method      string GetCimSessionComputerName()
GetCimSessionInstanceId     Method      guid GetCimSessionInstanceId()
GetHashCode                 Method      int GetHashCode()
GetObjectData               Method      void GetObjectData(System.Runtime.
Serializat...
GetType                     Method      type GetType()
ToString                    Method      string ToString()
Caption                     Property    string Caption {get;}
CommandLine                 Property    string CommandLine {get;}
CreationClassName           Property    string CreationClassName {get;}
CreationDate                Property    CimInstance#DateTime CreationDate {get;}
CSCreationClassName         Property    string CSCreationClassName {get;}
CSName                      Property    string CSName {get;}
Description                 Property    string Description {get;}
ExecutablePath              Property    string ExecutablePath {get;}
ExecutionState              Property    uint16 ExecutionState {get;}
Handle                      Property    string Handle {get;}
HandleCount                 Property    uint32 HandleCount {get;}
InstallDate                 Property    CimInstance#DateTime InstallDate {get;}
KernelModeTime              Property    uint64 KernelModeTime {get;}
MaximumWorkingSetSize       Property    uint32 MaximumWorkingSetSize {get;}
MinimumWorkingSetSize       Property    uint32 MinimumWorkingSetSize {get;}
Name                        Property    string Name {get;}
OSCreationClassName         Property    string OSCreationClassName {get;}
OSName                      Property    string OSName {get;}
OtherOperationCount         Property    uint64 OtherOperationCount {get;}
OtherTransferCount          Property    uint64 OtherTransferCount {get;}
PageFaults                  Property    uint32 PageFaults {get;}
PageFileUsage               Property    uint32 PageFileUsage {get;}
ParentProcessId             Property    uint32 ParentProcessId {get;}
PeakPageFileUsage           Property    uint32 PeakPageFileUsage {get;}
PeakVirtualSize             Property    uint64 PeakVirtualSize {get;}
PeakWorkingSetSize          Property    uint32 PeakWorkingSetSize {get;}
Priority                    Property    uint32 Priority {get;}
PrivatePageCount            Property    uint64 PrivatePageCount {get;}
ProcessId                   Property    uint32 ProcessId {get;}
PSComputerName              Property    string PSComputerName {get;}
QuotaNonPagedPoolUsage      Property    uint32 QuotaNonPagedPoolUsage {get;}
QuotaPagedPoolUsage         Property    uint32 QuotaPagedPoolUsage {get;}
QuotaPeakNonPagedPoolUsage  Property    uint32 QuotaPeakNonPagedPoolUsage {get;}
QuotaPeakPagedPoolUsage     Property    uint32 QuotaPeakPagedPoolUsage {get;}
ReadOperationCount          Property    uint64 ReadOperationCount {get;}
ReadTransferCount           Property    uint64 ReadTransferCount {get;}
SessionId                   Property    uint32 SessionId {get;}
Status                      Property    string Status {get;}
TerminationDate             Property    CimInstance#DateTime TerminationDate {get;}
ThreadCount                 Property    uint32 ThreadCount {get;}
UserModeTime                Property    uint64 UserModeTime {get;}
VirtualSize                 Property    uint64 VirtualSize {get;}
WindowsVersion              Property    string WindowsVersion {get;}
WorkingSetSize              Property    uint64 WorkingSetSize {get;}
WriteOperationCount         Property    uint64 WriteOperationCount {get;}
```

```
WriteTransferCount         Property       uint64 WriteTransferCount {get;}
Path                       ScriptProperty System.Object Path {get=$this.ExecutablePath;}
```

When the command runs without piping to the *Get-Member* object, we see that first the instance of the *Win32_UserAccount* WMI class returns. The output shows the user name, account type, SID, domain, and the caption of the user account. As shown in the output from *Get-Member*, a lot more information is available, but this is the default display. Following the user account information, the default process information displays the process ID, name, and a bit of performance information related to the processes associated with the user account:

```
PS C:\> $logon = Get-CimInstance win32_logonsession
PS C:\> Get-CimAssociatedInstance $logon

Name             Caption            AccountType       SID               Domain
----             -------            -----------       ---               ------
ed               IAMMRED\ed         512               S-1-5-21-14579... IAMMRED

ProcessId      : 2780
Name           : taskhostex.exe
HandleCount    : 215
WorkingSetSize : 8200192
VirtualSize    : 242356224


ProcessId      : 2804
Name           : rdpclip.exe
HandleCount    : 225
WorkingSetSize : 8175616
VirtualSize    : 89419776


ProcessId      : 2352
Name           : explorer.exe
HandleCount    : 1078
WorkingSetSize : 65847296
VirtualSize    : 386928640


ProcessId      : 984
Name           : powershell.exe
HandleCount    : 577
WorkingSetSize : 94527488
VirtualSize    : 690466816


ProcessId      : 296
Name           : conhost.exe
HandleCount    : 54
WorkingSetSize : 7204864
VirtualSize    : 62164992
```

If you do not want to retrieve both classes from the association query, you can specify the resulting class by name. To do this, use the resultclassname parameter from the *Get-CimAssociatedInstance* cmdlet. In the following example, only the *Win32_UserAccount* WMI class returns from the query:

```
PS C:\> $logon = Get-CimInstance win32_logonsession
PS C:\> Get-CimAssociatedInstance $logon -ResultClassName win32_useraccount

Name             Caption            AccountType       SID               Domain
----             -------            -----------       ---               ------
ed               IAMMRED\ed         512               S-1-5-21-14579... IAMMRED
```

When you work with the *Get-CimAssociatedInstance* cmdlet, the InputObject you supply must be a single instance. If you supply an object that contains more than one instance of the class, an error raises. This error appears in the following example, where more than one disk is provided to the InputObject parameter:

```
PS C:\> $disk = Get-CimInstance win32_logicaldisk
PS C:\> Get-CimAssociatedInstance $disk
Get-CimAssociatedInstance : Cannot convert 'System.Object[]' to the type
'Microsoft.Management.Infrastructure.CimInstance' required by parameter 'InputObject'.
Specified method is not supported.
At line:1 char:27
+ Get-CimAssociatedInstance $disk
+                           ~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-CimAssociatedInstance], Paramete
   rBindingException
    + FullyQualifiedErrorId : CannotConvertArgument,Microsoft.Management.Infrastructure.
   CimCmdlets.GetCimAssociatedInstanceCommand
```

There are two ways to correct this particular error. The first, and the easiest, is to use array indexing, as shown in the following example:

```
PS C:\> $disk = Get-CimInstance win32_logicaldisk
PS C:\> Get-CimAssociatedInstance $disk[0]

Name      PrimaryOwnerName Domain         TotalPhysicalMemory Model         Manufacturer
----      ---------------- ------         ------------------- -----         ------------
W8C504    ed               iammred.net    2147012608     Virtual Ma... Microsoft ...


PS C:\> Get-CimAssociatedInstance $disk[1]

Name             Hidden             Archive           Writeable         LastModified
----             ------             -------           ---------         ------------
c:\

NumberOfBlocks   : 265613312
BootPartition    : False
Name             : Disk #0, Partition #1
PrimaryPartition : True
Size             : 135994015744
Index            : 1
```

```
Domain              : iammred.net
Manufacturer        : Microsoft Corporation
Model               : Virtual Machine
Name                : W8C504
PrimaryOwnerName    : ed
TotalPhysicalMemory : 2147012608
```

Using array indexing is fine when you find yourself in the situation with an InputObject that contains an array. However, the results might be a bit inconsistent. A better approach is to ensure you do not have an array in the first place. To do this, use the filter parameter to reduce the number of instances of your WMI class that return. In the following example, the filter returns the number of WMI instances to the C drive:

```
PS C:\> $disk = Get-CimInstance win32_logicaldisk -Filter "name = 'c:'"
PS C:\> Get-CimAssociatedInstance $disk

Name             Hidden             Archive           Writeable         LastModified
----             ------             -------           ---------         ------------
c:\

NumberOfBlocks   : 265613312
BootPartition    : False
Name             : Disk #0, Partition #1
PrimaryPartition : True
Size             : 135994015744
Index            : 1


Domain              : iammred.net
Manufacturer        : Microsoft Corporation
Model               : Virtual Machine
Name                : W8C504
PrimaryOwnerName    : ed
TotalPhysicalMemory : 2147012608
```

An easy way to see the objects returned by the *Get-CimAssociatedInstance* cmdlet is to pipeline the returned objects to the *Get-Member* cmdlet and then to select the *typename* property. Because more than one instance of the object might return and clutter the output, it is important to choose unique typenames. The following example shows this command:

```
PS C:\> Get-CimAssociatedInstance $disk | gm | select typename -Unique

TypeName
--------
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_Directory
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_DiskPartition
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_ComputerSystem
```

Armed with this information, it is easy to explore the returned associations, as shown in the following example:

```
PS C:\> Get-CimAssociatedInstance $disk -ResultClassName win32_directory

Name             Hidden             Archive           Writeable         LastModified
----             ------             -------           ---------         ------------
c:\


PS C:\> Get-CimAssociatedInstance $disk -ResultClassName win32_diskpartition

Name             NumberOfBlocks BootPartition  PrimaryPartition Size            Index
----             -------------- -------------  ---------------  ----            -----
Disk #0, Part... 265613312      False          True             135994015744    1


PS C:\> Get-CimAssociatedInstance $disk -ResultClassName win32_Computersystem

Name      PrimaryOwnerName Domain        TotalPhysicalMemory Model         Manufacturer
----      ---------------  ------        ---------------     -----         ------------
W8C504    ed               iammred.net   2147012608          Virtual Ma... Microsoft ...
```

Keep in mind that the entire WMI class returns and is therefore ripe for further exploration. The easy way to do this is to store the results into a variable, and then walk through the data. Once you have what interests you, you might decide to display a nicely organized table, as shown in the following example:

```
PS C:\> $dp = Get-CimAssociatedInstance $disk -ResultClassName win32_diskpartition
PS C:\> $dp | FT deviceID, BlockSize, NumberOfBLicks, Size, StartingOffSet -AutoSize

deviceID              BlockSize NumberOfBLicks         Size StartingOffSet
--------              --------- --------------         ---- --------------
Disk #0, Partition #1       512                135994015744      368050176
```

Summary

This chapter discussed using CIM cmdlets to perform WMI discovery. We reviewed how to find WMI class methods as well as how to find classes by qualifier. Next, we covered how to return WMI information both locally and remotely. We concluded by examining WMI associations.

CHAPTER 10

Using the Windows PowerShell ISE

- Running the Windows PowerShell ISE
- Working with Windows PowerShell ISE snippets

The Windows PowerShell ISE in Windows PowerShell 3.0 is completely revised over its 2.0 cousin. The ISE in Windows PowerShell ISE stands for Integrated Scripting Environment, but that does not mean you must use it to write scripts. In fact, many IT pros like to use the Windows PowerShell ISE for interactive Windows PowerShell commands because it is easier to edit, has better tab completion, and has a built-in Command pane.

Running the Windows PowerShell ISE

On Windows 8, the Windows PowerShell ISE appears to be a bit hidden. In fact, on Windows Server 2012 it also is a bit hidden. On Windows Server 2012, a Windows PowerShell shortcut automatically appears on the desktop taskbar. Pinning Windows PowerShell to the Windows 8 desktop taskbar is also a Windows PowerShell best practice.

To start the Windows PowerShell ISE, you have a couple of choices. On the Start window of Windows Server 2012, you can type **PowerShell** and both Windows PowerShell and the Windows PowerShell ISE appear as search results. However, on Windows 8, this is not the case. You must type **PowerShell_ISE** to find the Windows PowerShell ISE. Another way to launch the Windows PowerShell ISE is to right-click the Windows PowerShell icon and choose either Windows PowerShell ISE or Run ISE as Administrator from the task menu. Figure 10-1 shows the task menu.

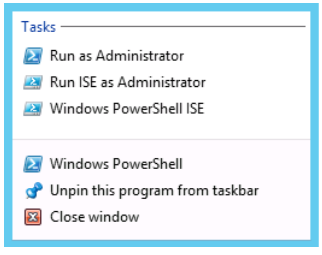


FIGURE 10-1 Right-clicking on the Windows PowerShell icon on the desktop taskbar displays a task menu that allows you to select the Windows PowerShell ISE.

In the Windows PowerShell console, you need to type only **ise** to launch the Windows PowerShell ISE. This shortcut permits quick access to the Windows PowerShell ISE when you need to type more than a few interactive commands.

Navigating the Windows PowerShell ISE

Once the Windows PowerShell ISE launches, two panes appear. On the left side of the screen is an interactive Windows PowerShell console. On the right side of the screen is the Command Add-on window. The Command Add-on is really a Windows PowerShell command explorer window. When you use the Windows PowerShell ISE in an interactive way, the Command Add-on gives you the ability to build a command by using the mouse. Once you have built the command, click the Run button to copy the command to the console window and execute the command. Figure 10-2 shows this view of the Windows PowerShell ISE.

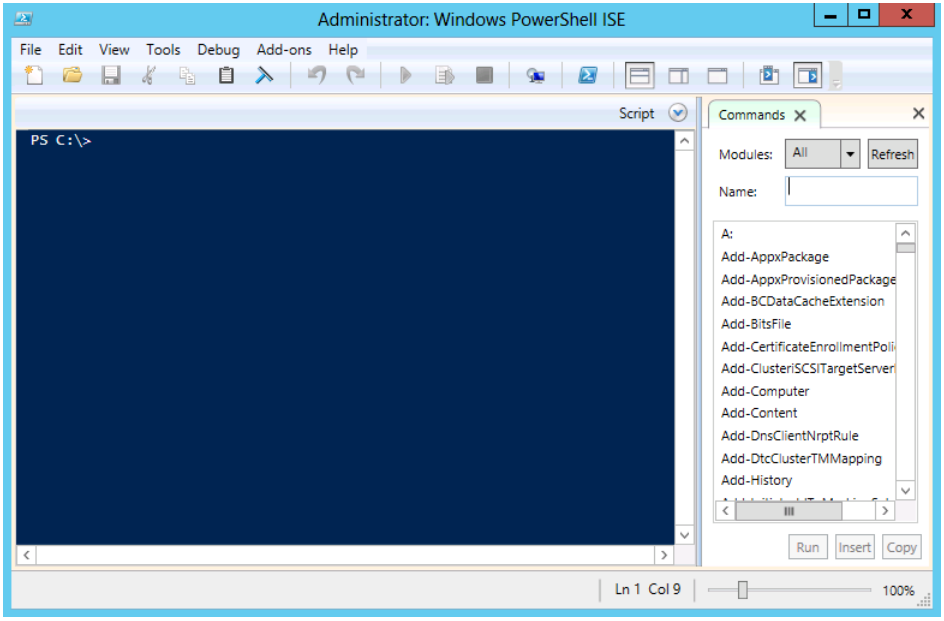


FIGURE 10-2 The Windows PowerShell ISE presents a Windows PowerShell console on the left side of the screen and a Command Add-on on the right side of the screen.

Typing into the Name text box causes the Command Add-on to search through all Windows PowerShell modules to retrieve a matching command. This is a great way to find and locate commands. By default, the Command Add-on uses a wildcard search pattern. Therefore, typing **wmi** returns five cmdlets that include that letter pattern, as shown in Figure 10-3.

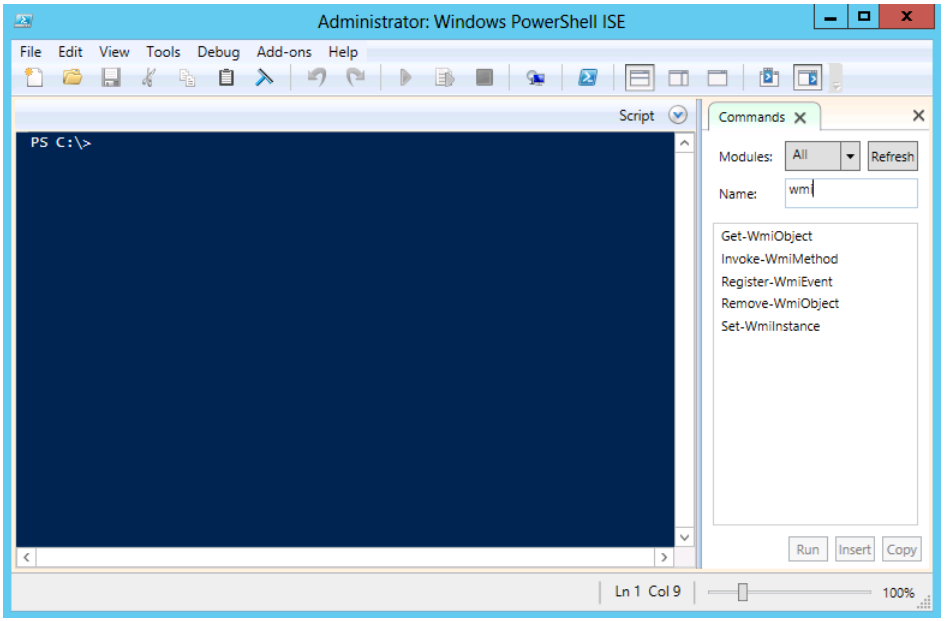


FIGURE 10-3 The Command Add-on uses a wildcard search pattern to find matching cmdlets.

Once you find the cmdlet that interests you, select it from the filter list of cmdlet names. Upon selection, the Command pane changes to the parameters for the selected cmdlet. Each parameter set appears on a different tab. Screen resolution really affects the usability of this feature. The greater the screen resolution, the more usable this feature becomes. With a small resolution, you have to scroll back and forth to see the parameter sets, and you have to scroll up and down to see the available parameters for a particular parameter set. In this view, it is easy to miss important parameters.

In Figure 10-4, the *Get-WmiObject* cmdlet queries the *Win32_Bios* WMI class. When you enter the WMI class name in the Class text box, the Run button executes the command. The Console pane displays the command first and then displays the output from running the command.

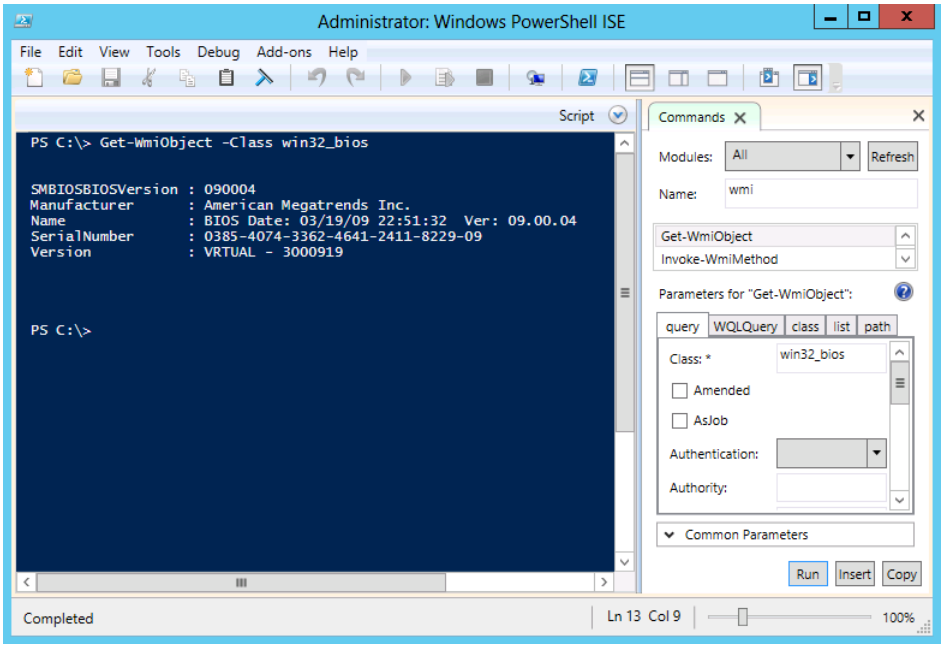


FIGURE 10-4 Select the command to run from the Command Add-on, fill in the required parameters, and press Run to execute Windows PowerShell cmdlets inside the Windows PowerShell ISE.

> ***NOTE*** **Using the Insert button inserts the command to the console but does not execute the command. This is great for occasions when you want to look over the command prior to actually executing it. It also provides you with the chance to edit the command prior to execution.**

To find and run commands through the Command Add-on, enter the command you are interested in running in the Name text box of the Command Add-on. Next, select the command from the filtered list. Enter the parameters in the Parameters for... parameter text box and press the Run button when finished. The results appear in the Command output pane in the center of the Windows PowerShell ISE.

Working with the Script pane

Pressing the Script arrow beside the word in the upper-right corner of the Console pane reveals a fresh Script pane. You can also obtain a fresh Script pane by selecting File | New or by clicking the small white piece of paper icon in the upper-left corner of the Windows PowerShell ISE. You can also use the keyboard shortcut Ctrl+N.

Just because it's called the Script pane does not mean that you have to enable script support to use it. As long as you do not save the script to a file, you can enter as complex of commands into the Script pane as you want. You can even run the script with the script execution policy restricted. Once you save the script to a file, however, the script execution policy comes into effect, and you will need to deal with the script execution policy at that point. Setting the Script Execution policy is discussed in Chapter 11, "Using Windows PowerShell Scripts."

You can still use the Command Add-on with the Script pane, but it requires an extra step. Use the Command Add-on as described in the preceding section, but instead of using the Run or Insert button, use the Copy button. Navigate to the appropriate section in the Script pane, and then use the Paste command by means of one of the following options: Select Paste from the right-click menu, select Paste from the Edit menu, click the Paste icon on the toolbar, or simply press Ctrl+V.

> ***NOTE*** **If you press the Insert button while the Script pane is maximized, the command is inserted into the hidden Console pane. Pressing Insert a second time inserts the command a second time on the same command line in the hidden Console pane. No notification that this occurs is presented.**

To run commands present in the Script pane, click the green triangle in the middle of the toolbar, press F-5, or select Run from the File menu. The commands from the Script pane transfer to the Console pane and then execute. Any output associated with the commands appears under the transferred commands. Once saved as a script, the commands no longer transfer to the Command pane. Rather, the path to the script appears in the Console pane along with any associated output.

You can continue to use the Command Add-on to build your commands as you pipeline the output from one cmdlet to another one. In Figure 10-5, the output from the *Get-WmiObject* cmdlet pipes to the *Format-Table* cmdlet. The properties chosen in the *Format-Table* cmdlet as well as the implementation of the Wrap switch are configured through the Command Add-on.

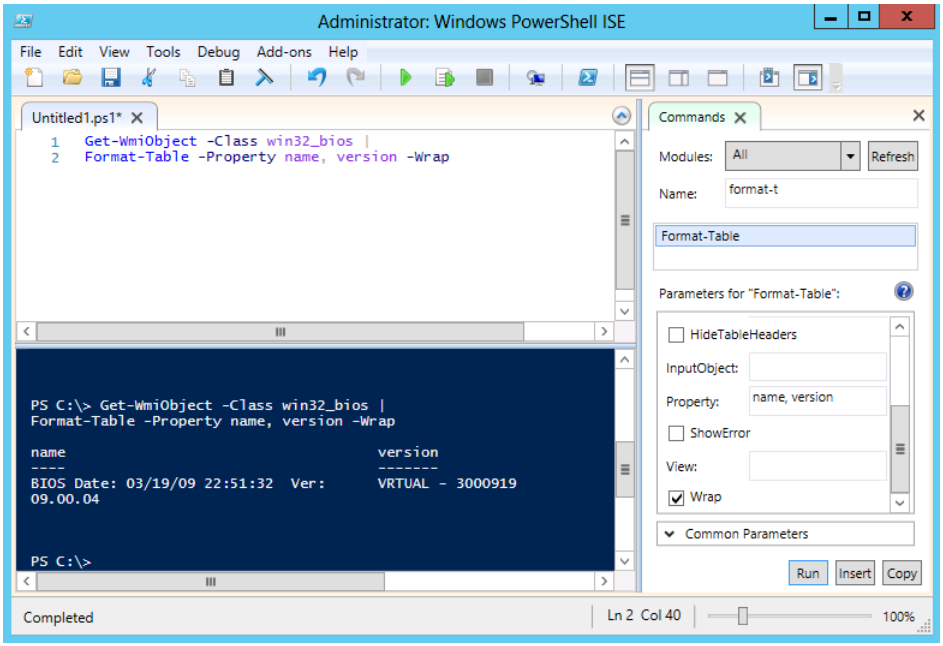


FIGURE 10-5 Use of the Command Add-on permits easy building of commands.

Tab expansion and Intellisense

Most advanced scripters will not use the Command Add-on because it consumes valuable screen real estate, plus it requires the use of the mouse to find and create commands. For advanced scripters, tab expansion and Microsoft Intellisense are the keys to productivity. To turn off the Command Add-on, either click the "x" in the upper-right corner of the Command Add-on or deselect Show Command Add-on from the View menu. Once deselected, the Windows PowerShell ISE remembers your preference and will not display the Command Add-on again until you re-select it.

Intellisense provides pop-up help and options permitting rapid command development without requiring complete syntax knowledge. When you type a cmdlet name, Intellisense supplies possible matches to the cmdlet names. Once you select the cmdlet, Intellisense displays the complete syntax of the cmdlet, as shown in Figure 10-6.

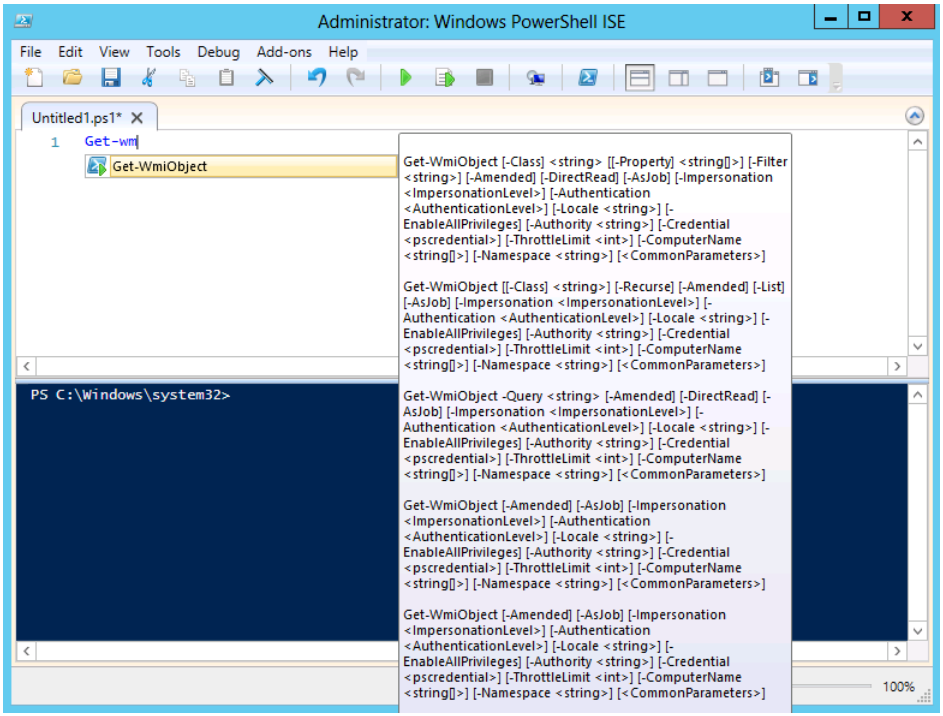


FIGURE 10-6 Once you select a particular cmdlet from the list, Intellisense displays the complete syntax.

When you select a particular cmdlet, as you come to parameters, Intellisense displays the applicable parameters in a list. Once Intellisense appears, use the Up and Down arrows to navigate within the list. Press Enter to insert the highlighted option. You can then fill in required values for parameters and go to the next parameter. Once again as you approach a parameter position, Intellisense displays the appropriate options in a list. This process continues until you complete the command. Figure 10-7 illustrates selecting the property parameter from the Intellisense list of optional parameters.

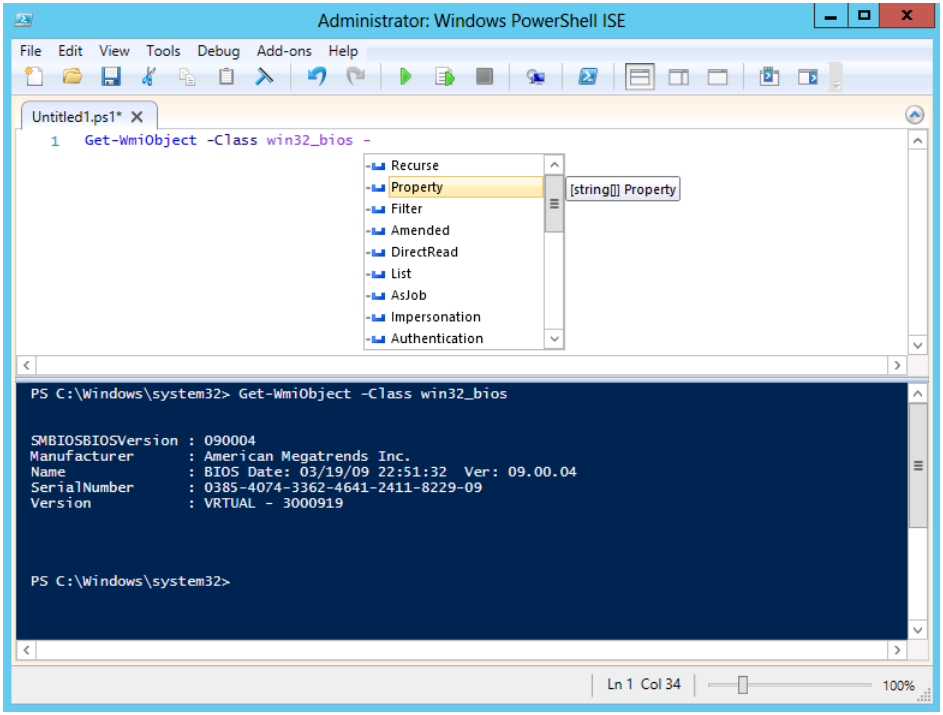


FIGURE 10-7 Intellisense displays parameters in a drop-down list. When you select a particular parameter, the data type of the property appears.

Working with Windows PowerShell ISE snippets

Even experienced scripters love to use Windows PowerShell ISE snippets because they are a great time saver. It takes just a little bit of familiarity with the snippets themselves, along with a bit of experience with the Windows PowerShell syntax. Once you have the requirements under your belt, you will be able to use the Windows PowerShell ISE snippets and create code faster than you previously believed was possible.

Using Windows PowerShell ISE snippets to create code

To start the Windows PowerShell ISE snippets, use the Ctrl+J keystroke combination (you can also use the mouse to select Edit | Start Snippets). Once the snippets appear, type the first letter of the snippet name to quickly jump to the appropriate portion of the snippets (you can also use the mouse to navigate up and down the snippet list). When you have identified the snippet you wish to use, press Enter to place the snippet at the current insertion point in your Windows PowerShell Script pane.

To create a new function through Windows PowerShell ISE snippets, press CTRL+J to start the Windows PowerShell ISE snippets. Next, type **f** to move to the "f" section of the Windows PowerShell ISE snippets. Use the Down arrow until you arrive at the simple function snippet. Press Enter to enter the simple function snippet into your code.

Creating new Windows PowerShell ISE snippets

After you spend a bit of time using Windows PowerShell ISE snippets, you will wonder how you ever managed to create script without them. In that same instant, you might also begin to think in terms of new snippets. Luckily, it is very easy to create a new Windows PowerShell ISE snippet. In fact, there is even a cmdlet to do this. The cmdlet is the *New-IseSnippet* cmdlet.

> ***NOTE*** **To create or use a user-defined Windows PowerShell ISE snippet, you must change the script execution policy to permit the execution of scripts. This is because user-defined snippets load from .xml files, and reading and loading files requires the script execution policy to permit running scripts. To verify your script execution policy, use the *Get-Execution-Policy* cmdlet. To set the script execution policy, use the *Set-ExecutionPolicy* cmdlet.**

Use the *New-IseSnippet* cmdlet to create a new Windows PowerShell ISE snippet. Once you create the snippet, it becomes immediately available in the Windows PowerShell ISE once you start the Windows PowerShell ISE snippets. The command syntax is simple, but the command takes a decent amount of space to complete. Only three parameters are required: *Description*, *Text*, and *Title*. The name of the snippet is the Title parameter. The snippet itself is typed into the Text parameter. When you want your code to appear on multiple lines, use the *`r* special character. Of course, to do this means your Text parameter must appear inside double quotation marks, not single quotation marks.

The following code creates a new Windows PowerShell ISE snippet that is a simplified Switch syntax. It is a single logical line of code:

```
New-IseSnippet -Title SimpleSwitch -Description "A simple switch statement" -Author "ed
wilson" -Text "Switch () `r{'param1' {  }`r}" -CaretOffset 9
```

Once you execute the *New-IseSnippet* command, it creates a new Snippets.xml file in the snippets directory within your WindowsPowerShell folder in your documents folder. Figure 10-8 shows the SimpleSwitch.snippets .xml file.

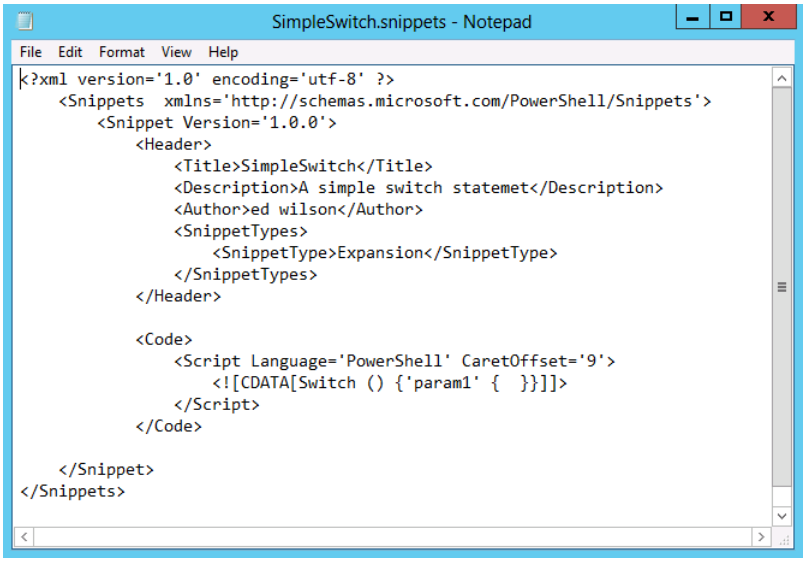


FIGURE 10-8 Windows PowerShell snippets store in a Snippets.xml file in your WindowsPowerShell folder.

User-defined snippets are permanent; that is, they survive closing and re-opening the Windows PowerShell ISE. They also survive reboots because they reside as .xml files in your WindowsPowerShell folder.

Removing user-defined Windows PowerShell ISE snippets

Although there is a *New-IseSnippet* cmdlet and a *Get-ISESnippet* cmdlet, there is no *Remove-ISESnippet* cmdlet. There is no need, really, because you have *Remove-Item*. To delete all your custom Windows PowerShell ISE snippets, use the *Get-ISESnippet* cmdlet to retrieve the snippets and the *Remove-Item* cmdlet to delete them. The following example shows this process:

```
Get-IseSnippet | Remove-Item
```

If you do not want to delete all your custom Windows PowerShell ISE snippets, use the *Where-Object* cmdlet to filter only the ones you do want to delete. The following example uses the *Get-ISESnippet* cmdlet to list all the user-defined Windows PowerShell ISE snippets on the system:

```
PS C:\Windows\system32> Get-IseSnippet


    Directory: C:\Users\administrator.IAMMRED\Documents\WindowsPowerShell\Snippets


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          7/1/2012   1:03 AM        653 bogus.snippets.ps1xml
-a---          7/1/2012   1:02 AM        653 mysnip.snippets.ps1xml
-a---          7/1/2012   1:02 AM        671 simpleswitch.snippets.ps1xml
```

Next, use the *Where-Object* cmdlet (*?* is an alias for the *Where-Object*) to return all the user-defined Windows PowerShell ISE snippets except the ones that contain the word Switch within the name. The snippets that make it through the filter are pipelined to the *Remove-Item* cmdlet. In the following example, the WhatIf switch shows which snippets will be removed by the command:

```
PS C:\Windows\system32> Get-IseSnippet | ? name -NotMatch 'switch' | Remove-Item -WhatIf
What if: Performing operation "Remove file" on Target "C:\Users\administrator.IAMMRED\
Documents\Win
dowsPowerShell\Snippets\bogus.snippets.ps1xml".
What if: Performing operation "Remove file" on Target "C:\Users\administrator.IAMMRED\
Documents\Win
dowsPowerShell\Snippets\mysnip.snippets.ps1xml".
```

Once you have confirmed that only the snippets you do not want to keep will be deleted, remove the WhatIf switch from the *Remove-Item* cmdlet and run the command a second time. To confirm which snippets remain, use the *Get-ISESnippet* cmdlet to see which Windows PowerShell ISE snippets are left on the system:

```
PS C:\Windows\system32> Get-IseSnippet | ? name -NotMatch 'switch' | Remove-Item

PS C:\Windows\system32> Get-IseSnippet


    Directory: C:\Users\administrator.IAMMRED\Documents\WindowsPowerShell\Snippets


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          7/1/2012   1:02 AM        671 simpleswitch.snippets.ps1xml
```

Summary

This chapter presented the advantages of using the Windows PowerShell ISE. The chapter provided an overview of the Windows PowerShell ISE, including the Script pane, tab expansion, and the Output pane. In addition, we covered the use of the Command Add-on. We concluded by demonstrating how to use snippets.

CHAPTER 11

Using Windows PowerShell scripts

- Writing Windows PowerShell scripts
- Scripting fundamentals
- Using the *While* statement
- Using the *Do…While* statement
- Using the *Do…Until* statement
- Using the *For* statement
- Using the *If* statement
- Using the *Switch* statement

With the ability to perform so many actions from inside Windows PowerShell in an interactive fashion, you might wonder, "Why do I need to write scripts?" For many network administrators, one-line Windows PowerShell commands will indeed solve many routine problems. This can become extremely powerful when the commands are combined into batch files and perhaps called from a log-on script. However, there are some very good reasons to write Windows PowerShell scripts. We will examine them in this chapter.

Why write Windows PowerShell scripts?

Perhaps the number one reason to write a Windows PowerShell script is to address recurring needs. As an example, consider the activity of producing a directory listing. The simple *Get-ChildItem* cmdlet does a good job, but after you decide to sort the listing and filter out only files of a certain size, you end up with the following command:

```
Get-ChildItem c:\fso | Where-Object Length -gt 1000 | Sort-Object -Property name
```

Even if you use tab completion, the previous command requires a bit of typing. One way to shorten it is to create a user-defined function. We will examine that technique later in this

chapter. For now, the easiest solution is to write a Windows PowerShell script. The following example shows the DirectoryListWithArguments.ps1 script:

```
DirectoryListWithArguments.ps1
foreach ($i in $args)
   {Get-ChildItem $i | Where-Object length -gt 1000 |
    Sort-Object -property name}
```

The DirectoryListWithArguments.ps1 script takes a single, unnamed argument that allows the script to be modified when it is run. This makes the script much easier to work with and adds flexibility.

An additional reason that network administrators write Windows PowerShell scripts is to run the script as a scheduled task. On Windows 8 and Windows Server 2012, you have two modules that assist in scheduling jobs: The ScheduledTasks module that permits working with the Windows job scheduler and the PSScheduledJob module that permits creating Windows PowerShell scheduled jobs. The PSScheduledJob module exists on any computer that runs Windows PowerShell 3.0.

The ListProcessesSortResults.ps1 script is a script that a network administrator might want to schedule to run several times a day. It produces a list of currently running processes and writes the results to a text file as a formatted and sorted table:

```
ListProcessesSortResults.ps1
$args = "localhost","loopback","127.0.0.1"

foreach ($i in $args)
   {$strFile = "c:\mytest\"+ $i +"Processes.txt"
    Write-Host "Testing" $i "please wait ...";
    Get-WmiObject -computername $i -class win32_process |
    Select-Object name, processID, Priority, ThreadCount, PageFaults, PageFileUsage |
    Where-Object {!$_.processID -eq 0} | Sort-Object -property name |
    Format-Table | Out-File $strFile}
```

Another reason for writing Windows PowerShell scripts is that it makes it easy to store and share both the "secret commands" and the ideas behind the scripts. For example, suppose you develop a script that will connect remotely to workstations on your network and search for user accounts that do not require a password. Obviously, an account without a password is a security risk! After some searching around, you discover the *WIN32_UserAccount* WMI class and develop a script that performs to your expectation. Because this is likely a script you will want to use on a regular basis, and perhaps share with other network administrators in your company, it makes sense to save it as a script. An example of such a script is AccountsWithNoRequiredPassword.ps1:

```
AccountsWithNoRequiredPassword.ps1
$args = "localhost"

foreach ($i in $args)
   {Write-Host "Connecting to" $i "please wait ...";
    Get-WmiObject -computername $i -class win32_UserAccount |
    Select-Object Name, Disabled, PasswordRequired, SID, SIDType |
    Where-Object {$_.PasswordRequired -eq 0} |
    Sort-Object -property name}
```

Scripting fundamentals

In its most basic form, a Windows PowerShell script is a collection of Windows PowerShell commands. For example, you can put the following command into a Windows PowerShell script and run it directly as it written:

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

To create a Windows PowerShell script, you only have to copy the command in a text file and save the file by using a .PS1 extension. If you create the file in the Windows PowerShell ISE and save the file, the .PS1 extension is added automatically. If you double-click the file, it will open in Notepad by default.

Running Windows PowerShell scripts

To run the script, you can open the Windows PowerShell console and drag the file to the console. If you first copy the path of the script to the Clipboard, you can later right-click inside the Windows PowerShell console to paste the path of the script into the console. Then you only need to press Enter to run the script. You just printed a string that represents the path of the script, as shown in the following example:

```
PS C:\> "C:\fso\test.ps1"
C:\fso\test.ps1
```

In Windows PowerShell, when you want to print a string in the console, you put it in quotation marks. You do not have to use *Wscript.Echo* or similar commands that you use in VBScript. It is easier and simpler, but it is something that takes getting used to. Ok, you figure out that you just displayed a string, so you remove the quotation marks and press Enter. This time, you receive a real error message. You might ask, "What now?" Figure 11-1 shows the error message that relates to the script execution policy that disallows the running of scripts.

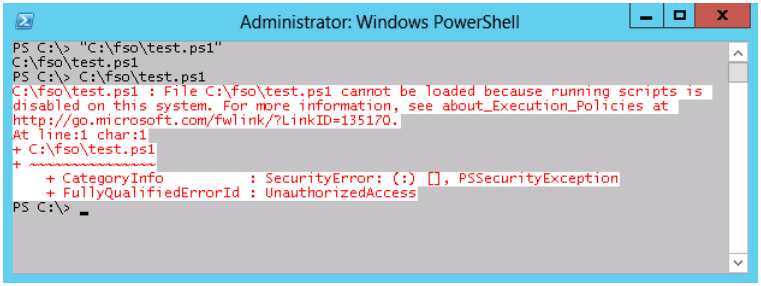


FIGURE 11-1 By default, an attempt to run a Windows PowerShell script generates an error message.

Enabling Windows PowerShell scripting support

By default, Windows PowerShell disallows the execution of scripts. Script support can be controlled by using Group Policy, but if it is not, and if you have administrator rights on your computer, you can use the *Set-ExecutionPolicy* Windows PowerShell cmdlet to turn on script support. The following list shows the six levels that can be enabled by using the *Set-ExecutionPolicy* cmdlet:

- **Restricted** Does not load configuration files or run scripts. "Restricted" is the default.
- **AllSigned** Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.
- **RemoteSigned** Requires that all scripts and configuration files downloaded from the Internet be signed by a trusted publisher.
- **Unrestricted** Loads all configuration files and runs all scripts. If you run an unsigned script that was downloaded from the Internet, you are prompted for permission before it runs.
- **Bypass** Nothing is blocked and there are no warnings or prompts.
- **Undefined** Removes the currently assigned execution policy from the current scope. This parameter will not remove an execution policy that is set in a Group Policy scope.

In addition to six levels of execution policy, there are three different scopes for the execution policies. The following list shows the three different execution policy scopes:

- **Process** The execution policy affects only the current Windows PowerShell process.
- **CurrentUser** The execution policy affects only the current user.
- **LocalMachine** The execution policy affects all users of the computer.

Setting the LocalMachine execution policy requires administrator rights on the local computer. By default, a non-elevated user has rights to set the script execution policy for the CurrentUser user scope that affects his own execution policy.

With so many choices available to you for a script execution policy, you might be wondering which one is appropriate for you. The Windows PowerShell team recommends the *RemoteSigned* setting, stating that it is appropriate for most circumstances. Remember that even though descriptions of the various policy settings use the term *Internet*, this may not always refer to the World Wide Web or even to locations outside your own firewall. This is because Windows PowerShell obtains its script origin information by using the Internet Explorer zone settings. This means anything that comes from a computer other than your own is in the Internet zone. You can change the Internet Explorer zone settings by using Internet Explorer, the registry, or Group Policy.

If you do not want to see the confirmation message when you change the script execution policy on Windows PowerShell 3.0, use the *-Force* parameter.

To view the execution policy for all scopes, use the List parameter when calling the *Get-ExecutionPolicy* cmdlet, as shown in the following example:

```
PS C:\> Get-ExecutionPolicy -List

                                Scope                         ExecutionPolicy
                                -----                         ---------------
                          MachinePolicy                               Undefined
                             UserPolicy                               Undefined
                                Process                               Undefined
                            CurrentUser                            RemoteSigned
                           LocalMachine                              Restricted
```

Use the *Set-ExecutionPolicy* cmdlet to change the script execution policy to *unrestricted*, as shown in the following example:

```
Set-ExecutionPolicy unrestricted
```

Once you set the execution policy, it is a good idea to use the *Get-ExecutionPolicy* cmdlet to retrieve the current effective script execution policy, as shown in the following example:

```
Get-ExecutionPolicy
```

The result prints to Windows PowerShell console:

```
Unrestricted
```

> ***TIP*** **If the execution policy on Windows PowerShell is set to *restricted*, how can you use a script to determine the execution policy? One method is to use the Bypass parameter when calling Windows PowerShell to run the script. The Bypass parameter bypasses the script execution policy for the duration of the script run when calling it.**

Transitioning from command line to script

Now that you have everything set up to enable script execution, you can run your StopNotepad.ps1 script:

```
StopNotepad.ps1
Get-Process Notepad | Stop-Process
```

If an instance of the Notepad process is running, everything is successful. However, if there is no instance of Notepad running, an error is generated:

```
Get-Process : Cannot find a process with the name 'Notepad'. Verify the process
 name and call the cmdlet again.
At C:\Documents and Settings\ed\Local Settings\Temp\tmp1DB.tmp.ps1:14 char:12
+ Get-Process  <<<< Notepad | Stop-Process
```

> ***NOTE*** **Be sure to read error messages when you use Windows PowerShell. It's a good habit to cultivate.**

The first part of the error message provides a description of the problem. In this example, it could not find a process with the name of *Notepad*. The second part of the error message shows the position in the code where the error occurred. This is known as the *position message*. The first line of the position message states the error occurred on line 14. The second portion has a series of arrows that point to the command that failed. The *Get-Process* cmdlet command is the one that failed:

```
At C:\Documents and Settings\ed\Local Settings\Temp\tmp1DB.tmp.ps1:14 char:12
+ Get-Process  <<<< Notepad | Stop-Process
```

The easiest way to eliminate this error message is to use the -ErrorAction parameter and specify the *SilentlyContinue* value. This is basically the same as using the *On Error Resume Next* command from VBScript. The really useful feature of the *-ErrorAction* parameter is that it can be specified on a cmdlet-by-cmdlet basis. In addition, there are four values that can be used. The following list shows the allowed values for the *-ErrorAction* parameter:

- *SilentlyContinue*
- *Continue* (the default value)
- *Inquire*
- *Stop*

In the StopNotepadSilentlyContinue.ps1 script, add the *-ErrorAction* parameter to the *Get-Process* cmdlet to skip any error that might arise if the Notepad process does not exist. To make the script easier to read, break the code at the pipeline character. The pipeline character is not the line continuation character. The backtick (`) character, also known as the grave character, is used when a line of code is too long and must be broken into two physical lines of code. The key thing to be aware of is that the two physical lines form a single logical line of code. The following example shows how to use line continuation:

```
Write-Host -foregroundcolor green "This is a demo " `
          "of the line continuation character"
```

The following example shows the StopNotepadSilentlyContinue.ps1 script:

```
StopNotepadSilentlyContinue.ps1
Get-Process -name Notepad -erroraction silentlycontinue |
Stop-Process
```

Because you are writing a script, you can take advantage of some features of a script. One of the first things you can do is use a variable to hold the name of the process to be stopped. This has the advantage of enabling you to easily change the script to allow for stopping of processes other than Notepad. All variables begin with the dollar sign (*$*). The following example shows the line that holds the name of the process in a variable:

```
$process= "notepad"
```

Another improvement to the script is one that provides information about the process that is stopped. The *Stop-Process* cmdlet returns no information when it is used. However, by using the *-PassThru* parameter, the process object is passed along in the pipeline. Use this parameter

and pipeline the process object to the *ForEach-Object* cmdlet. Use the *$_* automatic variable to refer to the current object on the pipeline and select the name and process ID of the process that is stopped. The concatenation operator in Windows PowerShell is the plus (+) sign. Use it to display the values of the selected properties in addition to the strings completing your sentence, as shown in the following example:

```
ForEach-Object { $_.name + ‘ with process ID: ‘ +  $_.ID + ‘ was stopped.’}
```

The following example shows the complete StopNotepadSilentlyContinuePassThru.ps1 script:

```
StopNotepadSilentlyContinuePassThru.ps1
$process = "notepad"
Get-Process -name $Process -erroraction silentlycontinue |
Stop-Process -passthru |
ForEach-Object { $_.name + ‘ with process ID: ‘ +  $_.ID + ‘ was stopped.’}
```

When you run the script with two instances of Notepad running, you get the following output:

```
notepad with process ID: 2088 was stopped.
notepad with process ID: 2568 was stopped.
```

An additional advantage of the StopNotepadSilentlyContinuePassThru.ps1 script is that you can use it to stop different processes. You can assign multiple process names (an array) to the *$process* variable, and when you run the script, each process will be stopped. The following example shows how to assign the Notepad and Calc processes to the *$process* variable:

```
$process= "notepad", "calc"
```

When you run the script, both processes are stopped:

```
calc with process ID: 3428 was stopped.
notepad with process ID: 488 was stopped.
```

You could continue changing your script. You could put the code in a function, write command-line Help, and change the script so that it accepts command-line input or even reads a list of processes from a text file. As soon as you move from the command line to script, such options suddenly become possible.

Running Windows PowerShell scripts

You cannot simply double-click on a Windows PowerShell script and have it run. You cannot type the name in the Run dialog box, either. In Windows PowerShell, you can run scripts if you have enabled the execution policy, but you need to type the entire path to the script you want to run and make sure you include the .ps1 extension.

If you need to run a script from outside Windows PowerShell, you need to type the full path to the script, but you must feed it as an argument to the PowerShell.exe program. In addition, you probably want to specify the *-NoExit* argument so you can read the output from the script, as shown in Figure 11-2.

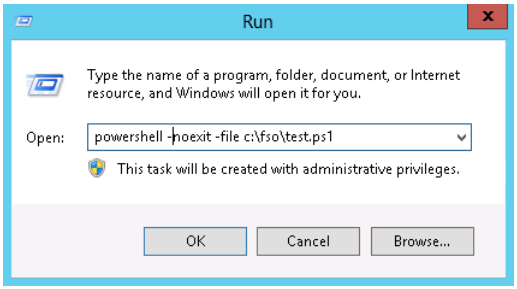


FIGURE 11-2 Use the *-NoExit* argument for the PowerShell.exe program to keep the console open after a script run.

TIP **Add a shortcut to Windows PowerShell in your SendTo folder. This folder is located in the Documents and Settings\%username% folder. When you create the shortcut, make sure you specify the *-NoExit* switch for PowerShell.exe or the output will scroll by so fast you will not be able to read it.**

Understanding variables and constants

Understanding the use of variables and constants in Windows PowerShell is fundamental to much of the flexibility of the Windows PowerShell scripting language. Variables are used to hold information for use later in the script. Variables can hold any type of data, including text, numbers, and even objects.

Using variables

By default when working with Windows PowerShell, you do not need to declare variables before use. When you use the variable to hold data, it is declared. All variable names must be preceded with a dollar sign (*$*). The following example illustrates creating a variable to hold the results from the *Get-Process* cmdlet. Once the process objects are stored in the *$process* variable, the contents of the variable are accessible. You can sort them and select specific properties:

```
PS C:\> $process = Get-Process

PS C:\> $process | sort cpu -Descending | select name, id, cpu -First 2 | ft -auto


Name           Id          CPU
----           --          ---
PindoraRadio 1324 1856.8019025
WINWORD      5184   99.5754383
```

There are a number of special variables in Windows PowerShell. These variables are created automatically and have a special meaning. As a best practice when writing scripts, do not create a variable with the same name (but with a different meaning) as these special variables. It can lead to unpredictable results and can be very difficult to troubleshoot. Table 11-1 shows a listing of the special variables and their associated meaning.

TABLE 11-1 Use of special variables

Name	Use
$	Contains the first token of the last line input into the shell.
$$	Contains the last token of the last line input into the shell.
$_	The current pipeline object; used in script blocks, filters, *Where-Object*, *ForEach-Object*, and *Switch*.
$?	Contains the success or fail status of the last statement.
$Args	Used in creating functions requiring parameters.
$Error	If an error occurs, the error object is saved in the *$error* variable.
$ExecutionContext	The execution objects available to cmdlets.
$foreach	Refers to the enumerator in a *ForEach* loop.
$HOME	The user's home directory; set to %HOMEDRIVE%\%HOMEPATH%.
$Input	Input is piped to a function or code block.
$Match	A hash table consisting of items found by the *-match* operator.
$MyInvocation	Information about the currently executing script or command line.
$PSHome	The directory where Windows PowerShell is installed.
$Host	Information about the currently executing host.
$LastExitCode	The exit code of the last native application to run.
$true	Boolean *true*.
$false	Boolean *false*.
$null	A null object.
$this	In the Types.ps1xml file and some script block instances, this represents the current object.
$OFS	Output Field Separator used when converting an array to a string.
$ShellID	Identifier for the shell. This value is used by the shell to determine the *ExecutionPolicy* and which profiles are run on startup.
$StackTrace	Contains detailed stack trace information about the last error.

Using the *While* statement

In VBScript, you have the *While…Wend* loop. An example of using the *While…Wend* loop is the WhileReadLineWend.vbs script. The first thing you do in the script is create an instance of the *FileSystemObject* and store it in the *objFSO* variable. You then use the *OpenTextFile* method to open a test file and store that object in the *objFile* variable. Next, you use the *While… Not…Wend* construction to read one line at a time from the text stream and display it on the screen. You continue to do this until you are at the end of the text stream object. A *While… Wend* loop continues to operate as long as a condition is evaluated as *true*. In this example, as long as you are not at the end of the stream, you will continue to read the line from the text file. The following example shows the WhileReadLineWend.VBS script:

```
WhileReadLineWend.vbs
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.OpenTextFile("C:\fso\testfile.txt")

While Not objFile.AtEndOfStream
 WScript.Echo objFile.ReadLine
Wend
```

Constructing the *While* statement

As you probably have already guessed, you have the same kind of construction available to you in Windows PowerShell that you do in VBScript. The While statement in Windows PowerShell is used in the same way that the *While…Wend* statement is used in VBScript.

In the DemoWhileLessThan.ps1 script, you first initialize the variable *$i* to be equal to *0*. You then use the While keyword to begin the While loop. In Windows PowerShell, you must include the condition that will be evaluated inside a set of parentheses. For this example, you determine the value of the *$i* variable with each pass through the loop. If the value of *$i* is less than the number *5*, you perform the action that is specified inside the braces (curly brackets) that set off the script block. In VBScript, the condition that is evaluated is positioned on the same line with the While statement, but no parentheses are required. Although this is convenient from a typing perspective, it actually makes the code a bit confusing to read. In Windows PowerShell, the statement is outside the parentheses and the condition is clearly delimited by the parentheses. In VBScript, the action that is performed is added between two words: While and Wend. In Windows PowerShell, there is no Wend statement, and the action to be performed is positioned inside a pair of braces. Although shocking at first to users coming from a VBScript background, the braces are always used to contain code. This is what is called a script block, and they are used everywhere. As soon as you are used to seeing them here, you also will find them with other language statements. One benefit is you do not have to look for items such as the Wend keyword or the *Loop* keyword (from the *Do…Loop* fame).

Understanding expanding strings

In\ Windows PowerShell, there are two kinds of strings: literal strings and expanding strings. In the DemoWhileLessThan.ps1 script, use the expanding string, which is signified by a double quotation mark ("). The literal string uses a single quotation mark ('). You should display the name of the variable, and you should display the value that is contained in the variable. This is a perfect place to showcase the expanding string. In an expanding string, the value that is contained in a variable is displayed to the screen when a line is evaluated. Consider the following example:

```
PS C:\> $i = 12
PS C:\> "$i is equal to $i"
12 is equal to 12
PS C:\>
```

As the example shows, you assign the value *12* to the variable *$i*. You then put *$i* inside a pair of double quotation marks, making an expanding string. When the line *"$i is equal to $i"* is evaluated, you obtain *"12 is equal to 12"*, which, while true, is barely illuminating.

Understanding literal strings

What you probably want to do is display both the name of the variable and the value that is contained inside it. In VBScript, you have to use concatenation. For this to work you have to use the literal string, as shown in the following example:

```
PS C:\> $i = 12
PS C:\> '$i is equal to ' + $i
$i is equal to 12
PS C:\>
```

If you want to use the advantage of the expanding string, you have to suppress the expanding nature of the expanding string for the first variable. To do this, use the escape character, which is the backtick (or grave character):

```
PS C:\> $i = 12
PS C:\> "`$i is equal to $i"
$i is equal to 12
PS C:\>
```

In the DemoWhileLessThan.ps1 script, use the expanding string to print your status message of the value of the *$i* variable during each trip through the *While* loop. Suppress the expanding nature of the expanding string for the first *$i* variable so you can see which variable you are using. As soon as you have done this, increment the value of the *$i* variable by one. To do this, use the *$i++* syntax. This is identical to the following example:

```
$i = $i + 1
```

The advantage is that the *$i*++ syntax is less typing. The following example shows the DemoWhileLessThan.ps1 script:

```
DemoWhileLessThan.ps1
$i = 0
While ($i -lt 5)
 {
  "`$i equals $i. This is less than  5"
  $i++
 } #end while $i lt 5
```

When you run the DemoWhileLessThan.ps1 script, you receive the following output:

```
$i equals 0. This is less than  5
$i equals 1. This is less than  5
$i equals 2. This is less than  5
$i equals 3. This is less than  5
$i equals 4. This is less than  5
PS C:\>
```

A practical example of using the *While* statement

Now that you know how to use the While loop, let's examine the WhileReadLine.ps1 script. The first step is to initialize the *$i* variable and set it equal to 0. Next, use the *Get-Content* cmdlet to read the contents of the testfile.txt and store the contents into the *$fileContents* variable.

Use the While statement to loop through the contents of the text file. Do this as long as the value of the *$i* variable is less than or equal to the number of lines in the text file. The number of lines in the text file is represented by the *length* property. Inside the script block, treat the contents of the *$fileContents* variable like it is an array (which it is), and use the *$i* variable to index into the array to print the value of each line in the *$fileContents* variable. Next, increment the value of the *$i* variable by one. The following example shows the WhileReadLine.ps1 script:

```
WhileReadLine.ps1
$i = 0
$fileContents = Get-Content -path C:\fso\testfile.txt
While ( $i -le $fileContents.length )
 {
  $fileContents[$i]
  $i++
 }
```

Using special features of Windows PowerShell

If you are thinking the WriteReadLine.ps1 script is a bit difficult, in reality it is about the same difficulty level as the VBScript version. The difference is you resorted to using arrays to work with the content you received from the *Get-Content* cmdlet. The VBScript version uses a

FileSystemObject and a *TextStreamObject* to work with the data. In reality, you do not have to use a script exactly like the WhileReadLine.ps1 script to read the contents of the text file. This is because the *Get-Content* cmdlet does this for you automatically. All you really have to do to display the contents of the TestFile.txt is use *Get-Content*, as shown in the following command:

```
Get-Content -path c:\fso\TestFile.txt
```

Because the results of the command are not stored in a variable, the contents are automatically emitted to the screen. You can further shorten the *Get-Content* command by using the *GC* alias (shortcut name for *Get-Content*) and by omitting the name of the *-Path* parameter (which is the default parameter). When you do this, you create a command that resembles the following:

```
GC c:\fso\TestFile.txt
```

To find the available aliases for the *Get-Content* cmdlet, use the *Get-Alias* cmdlet with the *-Definition* parameter. The *Get-Alias* cmdlet searches for aliases that have a definition that matches *Get-Content*. The following example shows the command and the output you receive:

```
PS C:\> Get-Alias -Definition Get-Content

CommandType     Name                            Definition
-----------     ----                            ----------
Alias           cat                             Get-Content
Alias           gc                              Get-Content
Alias           type                            Get-Content
```

In this section, you have seen that you can use the *While* statement in Windows PowerShell to perform looping. You have also seen that activities in VBScript that require looping do not always require you to use the looping behavior in Windows PowerShell because some cmdlets automatically display information. Finally, you saw how to find aliases for cmdlets you frequently use.

Using the *Do…While* statement

The *Do…While…Loop* statement is often used when working with VBScript. This section covers some of the advantages of the Do…While statement in Windows PowerShell.

The DemoDoWhile.vbs script illustrates using the *Do…While* statement in VBScript. The first step is to assign the value of *0* to the variable *i*. You then create an array. To do this, use the *Array* function and assign the numbers 1 through 5 to the variable *$ary*. Next, use the *Do…While…Loop* construction to walk through the array of numbers. As long as the value of the variable *i* is less than the number *5*, you display the value of the variable *i*. You then

increment the value of the variable and loop back around. The following example shows the DemoDoWhile.vbs script:

```
DemoDoWhile.vbs
i = 0
ary = Array(1,2,3,4,5)
Do While i < 5
 WScript.Echo ary(i)
 i = i + 1
Loop
```

When you run the DemoDoWhile.vbs script in Cscript at the command prompt, you see the numbers 1 through 5 displayed at the command prompt.

You can do exactly the same thing by using Windows PowerShell. The DemoDoWhile.ps1 script and the DemoDoWhile.vbs scripts are essentially the same. The differences between the two scripts are due to syntax differences between Windows PowerShell and VBScript. The first thing you do is assign the value of *1* to the variable *$i*. You then create an array of the numbers 1 through 5 and store that array in the *$ary* variable. You can use a shortcut in Windows PowerShell to make this a bit easier. Actually, arrays in Windows PowerShell are fairly easy anyway. If you want to create an array, you just have to assign multiple pieces of data to the variable. To do this, simply separate each piece of data by a comma:

```
$ary = 1,2,3,4,5
```

Using the *range* operator

If you want to create an array with 32,000 numbers in it, it would be impractical to type each number and separate it with a comma. In VBScript, you have to use a *For…Next…Loop* to add the numbers to the array. In Windows PowerShell, you can use the *range* operator. To do this, use a variable to hold the array of numbers that are created, and type the beginning and the ending number separated with two periods:

```
$ary = 1..5
```

Unfortunately, the *range* operator does not work for letters. But there is nothing to prevent you from creating a range of numbers that represent the ASCII value of each letter and then casting it to a string later.

Operating over an array

You are now ready for the *Do…While…Loop* in Windows PowerShell. First, use the *Do* statement and open a set of braces (curly brackets). A script block is inside these curly brackets. Next, index into the array. On your first pass through the array, the value of *$i* is equal to *0*. You therefore display the first element in the *$ary* array. Next, increment the value of the *$i* variable by one. You are now done with the script block, so look at the *While* statement. The condition you are examining is the value of the *$i* variable. As long as it is less than *5*, you will

continue to loop around. As soon as the value of *$i* is no longer less than the number *5*, you will stop looping. The following example shows this process:

```
DemoDoWhile.ps1
$i = 0
$ary = 1..5
do
{
 $ary[$i]
 $i++
} while ($i -lt 5)
```

One thing to be aware of, because it can be a bit confusing, is that you are evaluating the value of *$i*. You initialized *$i* at *0*. The first number in your array was *1*. But the first element number in the array is always *0* in Windows PowerShell (unlike VBScript which can start arrays with *0* or *1*). The While statement evaluates the value contained in the *$i* variable, not the value that is contained in the array. That is why you see the number *5* displayed.

Casting to ASCII values

You can change the DemoDoWhile.ps1 script and display the uppercase letters from A - Z. To do this, first initialize the *$i* variable and set it to *0*. Next, create a range of numbers from 65 through 91. These are the ASCII values for the capital letter A through the capital letter Z. Now, begin the *Do* statement and open your script block. To this point, the script is identical to the previous one. To obtain letters from numbers, cast the integer to a char. To do this, use the char data type and put it in square brackets. You then use this to convert an integer to an uppercase letter. To display the uppercase letter B from the ASCII value of *66*, the code should resemble the following:

```
PS C:\> [char]66
B
```

Because you know that the *$caps* variable contains an array of numbers ranging from 65 through 91, and the variable *$i* will hold numbers from 0 through 26, you index into the *$caps* array, cast the *integer* to a *char*, and display the results, as shown in the following example:

```
[char]$caps[$i]
```

You then increment the value of *$i* by one, close the script block, and enter the *While* statement where you check the value of *$i* to make sure it is less than *26*. As long as *$i* is less than *26*, you continue to loop around. The following example shows the complete DisplayCapitalLetters.ps1 script:

```
DisplayCapitalLetters.ps1
$i = 0
$caps = 65..91
do
{
 [char]$caps[$i]
 $i++
} while ($i -lt 26)
```

In this section, we explored the Do...While construction from Windows PowerShell by comparing it to a similar construction from VBScript. In addition, we examined the use of the *range* operator and casting.

Using the *Do…Until* statement

Looping technology is something that is essential to master. It occurs everywhere, and should be a tool you can use without thought. When you are confronted with a collection of items, an array, or other bundle of items, you have to know how to easily walk through the mess without resorting to research, panic, or hours searching the Internet with Bing search.

This section examines the *Do…Until…Loop* construction. Most of the scripts that do looping at the Microsoft TechNet Script Center seem to use the *Do…While…Loop*. The scripts that use *Do…Until…Loop* are typically used to read through a text file (do until the end of the stream) or to read through an Active X Data Object (ADO) recordset (do until the end of the file). As you will see here, these are not required coding conventions and are not meant to be limitations. You can frequently perform the same thing by using any of the different looping constructions.

Using the Windows PowerShell *Do…Loop* statement

You can write a looping script by using the *Do…Loop* statement in Windows PowerShell. In the DemoDoUntil.ps1 script, you first set the value of the *$i* variable to *0*. You then create an array with the numbers 1 through 5 in it and store that array in the *$ary* variable. You then arrive at the *Do…Loop* (Do-Until) construction. After the Do keyword, open a set of braces (curly brackets). Inside the curly brackets, use the *$i* variable to index into the *$ary* array and retrieve the value that is stored in the first element (element *0*) of the array. Next, increment the value of the *$i* variable by *1*. Continue to loop through the elements in the array until the value of the *$i* variable is equal to *5*. At that time, you end the script. This script resembles the DemoDoWhile.ps1 script examined in the preceding section. The following example shows this process:

```
DemoDoUntil.ps1
$i = 0
$ary = 1..5

Do
{
 $ary[$i]
 $i ++
} Until ($i -eq 5)
```

The *Do…While* and *Do…Until* statements always run once

In VBScript, if a *Do…While…Loop* condition is never *true*, the code inside the loop will never execute. In Windows PowerShell, the *Do…While* and the *Do…Until* constructions always run at least once. This can be unexpected behavior and is something you should focus on. This is illustrated in the DoWhileAlwaysRuns.ps1 script. The script assigns the value of *1* to the variable *$i*. Inside the script block for the *Do…While* loop, you print a message that states execution is taking place inside the *Do* loop. The loop condition continues while the variable *$i* is equal to *5*. As you can see, the value of the *$i* variable is *1*. Therefore, the value of the *$i* variable will never reach *5* because you are not incrementing it. The following example shows the DoWhileAlwaysRuns.ps1 script:

```
DoWhileAlwaysRuns.ps1
$i = 1

Do
{
 "inside the do loop"
} While ($i -eq 5)
```

When you run the script, the text "inside the do loop" is printed once.

What about a similar script that uses the *Do…Until* construction? The EndlessDoUntil.ps1 script is the same script as the DoWhileAlwaysRuns.ps1 script except for one small detail. Instead of using *Do…While*, you are using *Do…Until*. The rest of the script is the same. The value of the *$i* variable is equal to *1*, and in the script block for the *Do…Until* loop you print the string "inside the do loop." This line of code should execute once for each *Do* loop until the value of *$i* is equal to *5*. Because the value of *$i* is never increased to *5*, the script will continue to run. The following example shows the EndlessDoUntil.ps1 script:

```
EndlessDoUntil.ps1
$i = 1

Do
{
 "inside the do loop"
} Until ($i -eq 5)
```

Before you run the EndlessDoUntil.ps1 script, you should know how to interrupt the running of the script. To do this, hold down the CTRL key and press C (Ctrl+C). This is the same key stroke sequence that will break a runaway VBScript that is run in Cscript.

The While statement is used to prevent unwanted execution

If you have a situation where the script block must not execute if the condition is not *true*, you should use the While statement. We examined this statement in the "Using the *While* statement" section earlier in this chapter. Again, you have the same kind of script. You assign the value of *0* to the variable *$i*, but instead of using a Do … kind of construction, you use the While statement. The condition you are looking at is the same condition you used for the other scripts, where the value of *$i* is equal to *5*. Inside the script block, display a string that

states the process is inside the While loop. The following example shows the WhileDoesNotRun.ps1 script:

```
WhileDoesNotRun.ps1
$i = 0

While ($i -eq 5)
{
 "Inside the While Loop"
}
```

It is perhaps a bit anti-climactic, but go ahead and run the WhileDoesNotRun.ps1 script. There should be no output displayed to the console.

Using the *For* statement

In VBScript a *For…Next…Loop* is somewhat easy to create. An example of a simple *For…Next…Loop* is seen in DemoForLoop.vbs. You use the *For* keyword, define a variable to keep track of the count, indicate how far you will go, define your action, and do not forget to specify the *Next* keyword. That is about all there is to it. It sounds more difficult than it is. The following example shows DemoForLoop.vbs:

```
DemoForLoop.vbs
For i = 1 To 5
 WScript.Echo i
Next
```

Creating a *For…Loop*

The structure of the *For…Loop* in Windows PowerShell resembles the structure for VBScript. They both begin with the For keyword, they both initialize the variable, and they both specify how far the loop will progress. One difference is that a *For…Loop* in VBScript automatically increments the counter variable. In Windows PowerShell, the variable is not automatically incremented, and you add *$i++* to increment the *$i* variable by *1*. Inside the script block (braces, curly brackets), you display the value of the *$i* variable. The following example shows the DemoForLoop.ps1 script:

```
DemoForLoop.ps1
For($i = 0; $i -le 5; $i++)
{
 ‘$i equals ‘ + $i
}
```

The Windows PowerShell For statement is very flexible, and you can leave one or more elements of it out. In the DemoForWithoutInitOrRepeat.ps1 script, you exclude the first and last sections of the For statement. You set the *$i* variable equal to *0* on the first line of the script. Next, you come to the For statement. In the DemoForLoop.ps1 script, the *$i = 0* was moved from inside the For statement to the first line of the script. The semicolon is still required because it is

used to separate the three sections of the statement. The condition portion, *$i -le 5*, is the same as in the previous script. The repeat section, *$i ++*, is not used either.

In the script section of the For statement, you display the value of the *$i* variable, and you also increment the value of *$i* by one. There are two kinds of Windows PowerShell strings: expanding and literal. We examined these two types of strings earlier in the "Constructing the *While* statement" section of this chapter. In the DemoForLoop.ps1 script, you see an example of a literal string because what is entered is what is displayed:

```
'$i equals ' + $i
```

In the DemoForWithoutInitOrRepeat.ps1 script, you see an example of an expanding string. The value of the variable, not the variable name itself, is displayed. To suppress the expanding nature of the expanding string, escape the variable by using the backtick character. When you use the expanding string in this manner, it enables you to avoid concatenating the string and the variable as you did in the DemoForLoop.ps1 script. This following example shows this technique:

```
"`$i is equal to $i"
```

The value of *$i* must be incremented somewhere. Because it was not incremented in the repeat section of the *For* statement, you have to be able to increment it inside the script block. The following example shows the DemoForWithoutInitOrRepeat.ps1 script:

```
DemoForWithoutInitOrRepeat.ps1
$i = 0
For(;$i -le 5; )
{
 "`$i is equal to $i"
 $i++
}
```

When you run the DemoForWithoutInitOrRepeat.ps1 script, the output that is displayed resembles the output produced by the DemoForLoop.ps1. You would never be able to tell it was missing 2/3 of the parameters.

You can put your For statement into an infinite loop by omitting all three sections of the For statement. You must leave the semicolons as position holders. When you omit the three parts of the For statement, the For statement will resemble the following:

```
for(;;)
```

The ForEndlessLoop.ps1 script will create an endless loop, but you do not have to do this if this is not your desire. Instead, you can use an *If* statement to evaluate a condition and take action when the condition is met. We cover If statements in the "Using the *If* statement" section later in this chapter. In the ForEndlessLoop.ps1 script, you display the value of the *$i* variable and increment it by *1*. The semicolon is used to represent a new line. The For statement could therefore be written on three lines if you want to do this. This would be useful if you have a very complex For statement, as it would make the code easier to read. You can write

the script block for the ForEndlessLoop.ps1 script on different lines and exclude the semicolon, as shown in the following example:

```
{
 $i
 $i++
}
ForEndlessLoop.ps1
for(;;)
{
 $i ; $i++
}
```

When you run the ForEndlessLoop.ps1 script, you are greeted with a long line of numbers. To break out of the endless loop, press Ctrl+C at the Windows PowerShell prompt.

You can see that working with Windows PowerShell is all about choices. You can decide how you want to work and what you want to achieve. The For statement in Windows PowerShell is very flexible, and maybe one day you will find just the problem waiting for the solution that you have.

Using the *ForEach* statement

The *ForEach* statement resembles the *For…Each…Next* construction from VBScript. In the DemoForEachNext.vbs script, you create an array of five numbers. They number 1 through 5. You then use the *For…Each…Next* statement to walk your way through the array that is contained in the variable $*ary*. The variable *i* is used iterate through the elements of the array. The *For…Each* block is entered as long as there is at least one item in the collection or array. When the loop is entered, all statements inside the loop are executed for the first element. In the DemoForEachNext.vbs script, this means that the following command is executed for each element in the array:

```
Wscript.Echo i
```

As long as there are more elements in the collection or array, the statements inside the loop continue to execute for each element. When there are no more elements in the collection or array, the loop is exited, and execution continues with the statement following the *Next* statement. The following example shows this process in the DemoForEachNext.vbs script:

```
DemoForEachNext.vbs
ary = Array(1,2,3,4,5)
For Each i In ary
  WScript.Echo i
Next
Wscript.echo "All done"
```

The DemoForEachNext.vbs script works exactly like the DemoForEach.ps1 script. In the DemoForEach.ps1 Windows PowerShell script, you first create an array that contains the numbers 1 through 5 and store that array in the *$ary* variable:

```
$ary = 1..5
```

Then you use the ForEach statement to walk through the array contained in the *$ary* variable. Use the *$i* variable to keep track of your progress through the array. Inside the script block (the curly brackets), you display the value of each variable, as shown in the following DemoForEach.ps1 script:

```
DemoForEach.ps1
$ary = 1..5
Foreach ($i in $ary)
{
 $i
}
```

Using the *ForEach* statement from the Windows PowerShell console

The great advantage of Windows PowerShell is that you can also use the ForEach statement from inside the Windows PowerShell console, as shown in the following example:

```
PS C:\> $ary = 1..5
PS C:\> foreach($i in $ary) { $i }
1
2
3
4
5
```

The ability to use the *ForEach* statement from inside the Windows PowerShell console can give you excellent flexibility when you are working interactively. However, much of the work done at the Windows PowerShell console consists of using pipelining. When you are working with the pipeline, you can use the *ForEach-Object* cmdlet. This cmdlet behaves in a similar manner to the *ForEach* statement but is designed to handle pipelined input. The difference is that you do not have to use an intermediate variable to hold the contents of the array. You can create the array and send it across the pipeline. The other difference is that you do not have to create a variable to use for the enumerator. You can use the *$_* automatic variable (which represents the current item on the pipeline) instead, as shown in the following example:

```
PS C:\> 1..5 | ForEach-Object { $_ }
1
2
3
4
5
```

Exiting the *ForEach* statement early

Suppose you do not want to work with all the numbers in the array. In VBScript terms, leaving a *For…Each…Loop* early is called an Exit For statement. You have to use an If statement to perform the evaluation of the condition. When the condition is met, you call Exit For. In the DemoExitFor.vbs script, you use an inline If statement to make this determination. The inline syntax is more efficient for these kinds of things than spreading the statement across three different lines. The key thing to remember about the inline If statement is it does not conclude with the final End If statement. The following example shows the DemoExitFor.vbs script:

```
DemoExitFor.vbs
ary = Array(1,2,3,4,5)
For Each i In ary
 If i = 3 Then Exit For
 WScript.Echo i
Next
WScript.Echo "Statement following Next"
```

Using the *Break* statement

In Windows PowerShell terms, you use the *Break* statement to leave the loop early. Inside the script block, you use an If statement to evaluate the value of the *$i* variable. If it is equal to *3*, you call the Break statement and leave the loop:

```
if($i -eq 3) { break }
```

The following example shows the complete DemoBreakFor.ps1 script:

```
DemoBreakFor.ps1
$ary = 1..5
ForEach($i in $ary)
{
 if($i -eq 3) { break }
 $i
}
"Statement following foreach loop"
```

When the DemoBreakFor.ps1 script runs, it displays the numbers *1* and *2*. Then it leaves the *ForEach* loop and runs the line of code following the *ForEach* loop:

```
1
2
Statement following foreach loop
```

Using the Exit statement

If you do not want to run the line of code after the *Loop* statement, you would use the Exit statement instead of the *Break* statement. Keep in mind that when using Exit it will cause Windows PowerShell to close, unless you are running a script. If you use Exit in a function, the

Exit statement will close the Windows PowerShell console instead of exiting the function. This is shown in following example for the DemoExitFor.ps1 script:

```
DemoExitFor.ps1
$ary = 1..5
ForEach($i in $ary)
{
 if($i -eq 3) { exit }
 $i
}
"Statement following foreach loop"
```

When the DemoExitFor.ps1 script runs, the line of code following the ForEach loop never executes. This is because the Exit statement ends the script. Following are the results of running the DemoExitF0r.ps1 script:

```
1
2
```

You could achieve the same thing in VBScript by using the *Wscript.Quit* statement instead of Exit For. As with the DemoExitFor.ps1 script, the DemoQUitFor.vbs script never comes to the line of code following the *For…Each…Loop*, as shown in the DemoQuitFor.vbs script:

```
DemoQuitFor.vbs
ary = Array(1,2,3,4,5)
For Each i In ary
 If i = 3 Then WScript.Quit
 WScript.Echo i
Next
WScript.Echo "Statement following Next"
```

In this section, we examined the use of the ForEach statement. It is used when you do not know how many items are contained within a collection. It allows you to walk through the collection and to work with items from that collection on an individual basis. In addition, we examined two techniques for exiting a ForEach statement.

Using the *If* statement

In VBScript the *If…Then…End If* statement was somewhat straightforward. There were several rules to be aware of:

- The *If* and *Then* statements must be on the same line.
- The *If…Then…End If* statement must conclude with *End If*.
- *End If* is two words, not one word.

The following example shows the VBScript *If…Then…End If* statement in the DemoIf.vbs script:

```
DemoIf.vbs
a = 5
If a = 5 Then
 WScript.Echo "a equals 5"
End If
```

In the Windows PowerShell version of the *If…Then…End* If statement, there is no Then keyword, nor is there an End If statement. The Windows PowerShell If statement is easier to type. This simplicity, however, comes with a bit of complexity. The condition that is evaluated in the If statement is positioned in a set of smooth parentheses. In the DemoIf.ps1 script, you are checking whether the variable *$a* is equal to *5*, as shown in the following example:

```
If ($a -eq 5)
```

The code that is executed when the condition is *true* is positioned inside a pair of braces (curly brackets). Code inside a pair of curly brackets is called a script block in Windows PowerShell, and script blocks are everywhere. The following example shows the script block for the DemoIf.ps1 script:

```
{
   '$a equals 5'
}
```

The Windows PowerShell version of the DemoIf.vbs script is the DemoIf.ps1 script:

```
DemoIf.ps1
$a = 5
If($a -eq 5)
 {
   '$a equals 5'
 }
```

The one thing that is different about the Windows PowerShell If statement is the comparison operators. In VBScript the equal sign (=) is used as an assignment operator. It is also used as an equality operator for comparison. On the first line of code, the variable *a* is assigned the value *5*. This uses the equal sign as an assignment. On the next line of code, the *If* statement is used to see whether the value of *a* is equal to the number *5*. On this line of code, the equal sign is used as the equality operator:

```
a = 5
If a = 5 Then
```

In simple examples such as this, it is fairly easy to tell the difference between an equality operator and an assignment operator. In more complex scripts, however, things could be confusing. Windows PowerShell removes that confusion by having special comparison operators. It might help to realize the main operators are two letters long. Table 11-2 shows the comparison operators.

TABLE 11-2 Comparison operators

Operator	Description	Example	Result
-eq	equals	`$a = 5 ; $a -eq 4`	*False*
-ne	not equal	`$a = 5 ; $a -ne 4`	*True*
-gt	greater than	`$a = 5 ; $a -gt 4`	*True*
-ge	greater than or equal to	`$a = 5 ; $a -ge 5`	*True*
-lt	less than	`$a = 5 ; $a -lt 5`	*False*
-le	less than or equal to	`$a = 5 ; $a -le 5`	*True*
-like	wildcard comparison	`$a = "This is Text" ; $a -like "Text"`	*False*
-notlike	wildcard comparison	`$a = "This is Text" ; $a -notlike "Text"`	*True*
-match	regular expression comparison	`$a = "Text is Text" ; $a -match "Text"`	*True*
-notmatch	regular expression comparison	`$a = "This is Text" ; $a -notmatch "Text$"`	*False*

Using assignment and comparison operators

Any value assignment will evaluate to *true*, and therefore the script block is executed. In the following example, you assign the value *1* to the variable *$a*. In the condition for the *If* statement, you assign the value of *12* to the variable *$a*. Any assignment evaluates to *true*, and the script block executes:

```
PS C:\> $a = 1 ; If ($a = 12) { "its true" }
its true
```

Rarely do you test a condition and perform an outcome. Most of the time, you have to perform one action if the condition is *true*, and another action if the condition is *false*. In VBScript you used the *If…Else…End If* construction. The *Else* clause went immediately after the first outcome to be performed if the condition were *true*. This is seen in the DemoIfElse.vbs script:

```
DemoIfElse.vbs
a = 4
If a = 5 Then
 WScript.Echo "a equals 5"
Else
 WScript.Echo "a is not equal to 5"
End If
```

In Windows PowerShell, the syntax is not surprising. Following the closing curly brackets from the If statement script block, you add the Else keyword and open a new script block to hold the alternative outcome:

```
demoIfElse.ps1
$a = 4
If ($a -eq 5)
{
 ‘$a equals 5’
}
Else
{
 ‘$a is not equal to 5’
}
```

Things become confusing with VBScript when you want to evaluate multiple conditions and have multiple outcomes. The *Else If* clause provides for the second outcome. You have to evaluate the second condition. The *Else If* clause receives its own condition, which is followed by the *Then* keyword. Following the *Then* keyword, you list the code you want to execute. This is followed by the *Else* keyword and a pair of *End If* statements. This is seen in the DemoIfElseIfElse.vbs script:

```
DemoIfElseIfElse.vbs
a = 4
If a = 5 Then
 WScript.Echo "a equals 5"
Else If a = 3 Then
 WScript.Echo "a equals 3"
Else
 WScript.Echo "a does not equal 3 or 5"
End If
End If
```

Evaluating multiple conditions

The Windows PowerShell demoIfElseIfElse.ps1 script is a bit easier to understand because it avoids the double *End If* kind of scenario. For each condition you want to evaluate, use ElseIf (be aware that it is a single word). Put the condition inside a pair of smooth parentheses and open your script block. The following example shows the demoIfElseIfElse.ps1 script:

```
demoIfElseIfElse.ps1
$a = 4
If ($a -eq 5)
{
 ‘$a equals 5’
}
ElseIf ($a -eq 3)
{
 ‘$a is equal to 3’
}
Else
{
 ‘$a does not equal 3 or 5’
}
```

In this section, we examined the use of the If statement. We also covered comparison operators and assignment operators.

Using the *Switch* statement

As a best practice, you generally avoid using the ElseIf type of construction from either VBScript or Windows PowerShell because there is a better way to write the same code.

In VBScript, use the *Select Case* statement to evaluate a condition and select one outcome from a group of potential statements. In the DemoSelectCase.VBS script, the value of the variable *a* is assigned the value of *2*. The *Select Case* statement is used to evaluate the value of the variable *a*. The following example shows the syntax:

```
Select Case testexpression
```

The test expression that is evaluated is the variable *a*. Each of the different cases contains potential values for the test expression. If the value of the variable *a* is equal to *1*, the code *Wscript.Echo "a = 1"* is executed:

```
Case 1
   WScript.Echo "a = 1"
```

Each of the different cases is evaluated in the same manner. The *Case Else* expression is run if none of the previous expressions evaluate to *true*. The following example shows the complete DemoSelectCase.vbs script:

```
DemoSelectCase.vbs
a = 2
Select Case a
 Case 1
   WScript.Echo "a = 1"
 Case 2
   WScript.Echo "a = 2"
 Case 3
   WScript.Echo "a = 3"
 Case Else
   WScript.Echo "unable to determine value of a"
End Select
WScript.Echo "statement after select case"
```

Using the basic *Switch* statement

In Windows PowerShell, there is no *Select Case* statement. There is, however, the *Switch* statement. The *Switch* statement is the most powerful statement in the Windows PowerShell language. The *Switch* statement begins with the *Switch* keyword, and the condition to be evaluated is positioned inside a pair of smooth parentheses:

```
Switch ($a)
```

Next a pair of braces (curly brackets) is used to mark off the script block for the Switch statement. Inside the script block, each condition to be evaluated begins with a value followed by the script block to be executed in the event the value matches the condition:

```
1 { '$a = 1' }
2 { '$a = 2' }
3 { '$a = 3' }
```

Defining the Default condition

If no match is found, and the Default statement is not used, the *Switch* statement exits and the line of code that follows the *Switch* statement is executed. The Default statement performs a function similar to the one performed by the *Case Else* statement from the *Select Case* statement. The following example shows the Default statement:

```
Default { 'unable to determine value of $a' }
```

The following example shows the complete DemoSwitchCase.ps1 script:

```
DemoSwitchCase.ps1
$a = 2
Switch ($a)
{
 1 { '$a = 1' }
 2 { '$a = 2' }
 3 { '$a = 3' }
 Default { 'unable to determine value of $a' }
}
"Statement after switch"
```

Understanding matching

With the *Select Case* statement, the first matching case is the one that is executed. As soon as that code executes, the line following the *Select Case* statement is executed. If the condition matches multiple cases in the *Select Case* statement, only the first match in the list is executed. Matches from lower in the list are not executed. Therefore, make sure that the most desirable code to execute is positioned highest in the *Select Case* order.

With the *Switch* statement in Windows PowerShell, order is not a major design concern. This is because every match from inside the *Switch* statement will be executed. An example of this is in the DemoSwitchMultiMatch.ps1 script:

```
DemoSwitchMultiMatch.ps1
$a = 2
Switch ($a)
{
 1 { ‘$a = 1’ }
 2 { ‘$a = 2’ }
 2 { ‘Second match of the $a variable’ }
 3 { ‘$a = 3’ }
 Default { ‘unable to determine value of $a’ }
}
"Statement after switch"
```

When the DemoSwitchMultiMatch.ps1 script runs, the second condition and third condition will both be matched, and therefore their associated script blocks are executed. The DemoSwitchMultiMatch.ps1 script produces the output in the following example:

```
$a = 2
Second match of the $a variable
Statement after switch
```

Evaluating an array

If an array is stored in the variable *a* in the DemoSelectCase.vbs script, a type mismatch error will be produced, as shown in the following example:

```
Microsoft VBScript runtime error: Type mismatch
```

The Windows PowerShell *Switch* statement can handle an array in the variable *$a* without any modification:

```
$a = 2,3,5,1,77
```

The following example shows the complete DemoSwitchArray.ps1 script:

```
DemoSwitchArray.ps1
$a = 2,3,5,1,77
Switch ($a)
{
 1 { ‘$a = 1’ }
 2 { ‘$a = 2’ }
 3 { ‘$a = 3’ }
 Default { ‘unable to determine value of $a’ }
}
"Statement after switch"
```

Controlling matching behavior

If you do not want the multi-match behavior of the Switch statement, you can use the Break statement to change the behavior. In the DemoSwitchArrayBreak.ps1 script, the Switch statement will be exited when the first match occurs because each of the match condition script blocks contains the Break statement, as shown in the following example:

```
 1 { '$a = 1' ; break }
 2 { '$a = 2' ; break }
 3 { '$a = 3' ; break }
```

You are not required to include the Break statement with each condition; instead, you could use it to exit the switch only after a particular condition is matched. The following example shows the complete DemoSwitchArrayBreak.ps1 script:

```
DemoSwitchArrayBreak.ps1
$a = 2,3,5,1,77
Switch ($a)
{
 1 { '$a = 1' ; break }
 2 { '$a = 2' ; break }
 3 { '$a = 3' ; break }
 Default { 'unable to determine value of $a' }
}
"Statement after switch"
```

In this section, we examined the use of the Windows PowerShell Switch statement. We also discussed the matching behavior of the Switch statement and use of Break statement.

Summary

This chapter covered the fundamentals of Windows PowerShell scripting. The chapter began with a discussion of the reasons to write scripts. Next, we examined how to run scripts and set the script execution policy. Finally, we reviewed the elements of Windows PowerShell scripts such as variables and the various language statements.

CHAPTER 12

Working with functions

- Understanding functions
- Using multiple input parameters
- Using functions to encapsulate business logic
- Using functions to provide ease of modification

There are clear-cut guidelines that you can use to design script. You can use these guidelines to ensure that scripts are easy to understand, easy to maintain, and easy to troubleshoot. In this chapter, we examine the reasons for scripting guidelines and provide examples of both good code and bad code design.

Understanding functions

In Windows PowerShell, functions have moved to the forefront as the primary programming element used when writing Windows PowerShell scripts. This is not necessarily due to improvements in functions per se, but rather a combination of factors including the maturity of Windows PowerShell scriptwriters. In Windows PowerShell 1.0, functions were not well understood, perhaps due to the lack of clear documentation as to their use, purpose, and application.

To create a function, you begin with the Function keyword followed by the name of the function. As a best practice, use the Windows PowerShell verb and noun combination when you create functions. Choose the verb from the standard list of Windows PowerShell verbs to make your functions easier to remember. It is a best practice to avoid creating new verbs when there is an existing verb that can easily do the job.

You can get an idea of the verb coverage by using the *Get-Command* cmdlet and piping the results to the *Group-Object* cmdlet, as shown in the following example:

```
Get-Command -CommandType cmdlet | Group-Object -Property Verb |
Sort-Object -Property count -Descending
```

The following example shows the output after the preceding command is run:

```
Count Name                      Group
----- ----                      -----
   98 Get                       {Get-Acl, Get-Alias, Get-AppLockerFileInformation...
   48 Set                       {Set-Acl, Set-Alias, Set-AppLockerPolicy, Set-Aut...
   38 New                       {New-Alias, New-AppLockerPolicy, New-CertificateN...
   31 Remove                    {Remove-AppxPackage, Remove-AppxProvisionedPackag...
   15 Add                       {Add-AppxPackage, Add-AppxProvisionedPackage, Add...
   11 Invoke                    {Invoke-BpaModel, Invoke-CimMethod, Invoke-Comman...
   11 Import                    {Import-Alias, Import-Certificate, Import-Clixml,...
   11 Export                    {Export-Alias, Export-Certificate, Export-Clixml,...
   10 Test                      {Test-AppLockerPolicy, Test-Certificate, Test-Com...
   10 Enable                    {Enable-ComputerRestore, Enable-JobTrigger, Enabl...
   10 Disable                   {Disable-ComputerRestore, Disable-JobTrigger, Dis...
    9 Clear                     {Clear-Content, Clear-EventLog, Clear-History, Cl...
    8 Start                     {Start-BitsTransfer, Start-DtcDiagnosticResourceM...
    8 Write                     {Write-Debug, Write-Error, Write-EventLog, Write-...
    7 Out                       {Out-Default, Out-File, Out-GridView, Out-Host...}
    6 ConvertTo                 {ConvertTo-Csv, ConvertTo-Html, ConvertTo-Json, C...
    6 Register                  {Register-CimIndicationEvent, Register-EngineEven...
    6 Stop                      {Stop-Computer, Stop-DtcDiagnosticResourceManager...
    5 Format                    {Format-Custom, Format-List, Format-SecureBootUEF...
    4 Update                    {Update-FormatData, Update-Help, Update-List, Upd...
    4 Unregister                {Unregister-Event, Unregister-PSSessionConfigurat...
    4 Show                      {Show-Command, Show-ControlPanelItem, Show-EventL...
    4 ConvertFrom               {ConvertFrom-Csv, ConvertFrom-Json, ConvertFrom-S...
    3 Receive                   {Receive-DtcDiagnosticTransaction, Receive-Job, R...
    3 Wait                      {Wait-Event, Wait-Job, Wait-Process}
    3 Complete                  {Complete-BitsTransfer, Complete-DtcDiagnosticTra...
    3 Select                    {Select-Object, Select-String, Select-Xml}
    3 Resume                    {Resume-BitsTransfer, Resume-Job, Resume-Service}
    3 Suspend                   {Suspend-BitsTransfer, Suspend-Job, Suspend-Service}
    3 Rename                    {Rename-Computer, Rename-Item, Rename-ItemProperty}
    2 Restore                   {Restore-Computer, Restore-IscsiVirtualDisk}
    2 Resolve                   {Resolve-DnsName, Resolve-Path}
    2 Restart                   {Restart-Computer, ReStart-Service}
    2 Save                      {Save-Help, Save-WindowsImage}
    2 Send                      {Send-DtcDiagnosticTransaction, Send-MailMessage}
    2 Disconnect                {Disconnect-PSSession, Disconnect-WSMan}
    2 Dismount                  {Dismount-IscsiVirtualDiskSnapshot, Dismount-Wind...
    2 Connect                   {Connect-PSSession, Connect-WSMan}
    2 Checkpoint                {Checkpoint-Computer, Checkpoint-IscsiVirtualDisk}
    2 Move                      {Move-Item, Move-ItemProperty}
    2 Mount                     {Mount-IscsiVirtualDiskSnapshot, Mount-WindowsImage}
    2 Measure                   {Measure-Command, Measure-Object}
    2 Join                      {Join-DtcDiagnosticResourceManager, Join-Path}
    2 Install                   {Install-NfsMappingStore, Install-WindowsFeature}
    2 Unblock                   {Unblock-File, Unblock-Tpm}
    2 Convert                   {Convert-IscsiVirtualDisk, Convert-Path}
    2 Undo                      {Undo-DtcDiagnosticTransaction, Undo-Transaction}
    2 Copy                      {Copy-Item, Copy-ItemProperty}
    2 Use                       {Use-Transaction, Use-WindowsUnattend}
    1 Tee                       {Tee-Object}
    1 Trace                     {Trace-Command}
```

```
  1 Uninstall               {Uninstall-WindowsFeature}
  1 Switch                  {Switch-Certificate}
  1 Compare                 {Compare-Object}
  1 Repair                  {Repair-WindowsImage}
  1 Sort                    {Sort-Object}
  1 Reset                   {Reset-ComputerMachinePassword}
  1 Confirm                 {Confirm-SecureBootUEFI}
  1 Read                    {Read-Host}
  1 Push                    {Push-Location}
  1 Where                   {Where-Object}
  1 Limit                   {Limit-EventLog}
  1 Initialize              {Initialize-Tpm}
  1 Group                   {Group-Object}
  1 ForEach                 {ForEach-Object}
  1 Expand                  {Expand-IscsiVirtualDisk}
  1 Exit                    {Exit-PSSession}
  1 Enter                   {Enter-PSSession}
  1 Debug                   {Debug-Process}
  1 Split                   {Split-Path}
  1 Pop                     {Pop-Location}
```

The command was run on Windows Server 2012 and includes cmdlets from the default modules. As shown in the example, *Get* is used the most by the default cmdlets, followed distantly by *Set*, *New*, and *Remove*.

A function is not required to accept any parameters. In fact, many functions do not require input to perform their job in the script. Let's use an example to illustrate this point. A common task for network administrators is obtaining the operating system version. Scriptwriters often need to do this to ensure their script uses the correct interface or exits gracefully. It is also quite common that one set of files is copied to a desktop running one version of the operating system, and a different set of files is copied for another version of the operating system.

The first step to create a function is to decide on a name. Because the function is going to retrieve information, the best verb to use from the listing of cmdlet verbs in the preceding example is *Get*. For the noun portion of the name, it is best to use something that describes the information that will be obtained. In our example, the noun *OperatingSystemVersion* makes sense. An example of such a function is the function in the Get-OperatingSystemVersion.ps1 script. The *Get-OperatingSystemVersion* function uses WMI to obtain the version of the operating system. In this, the most basic form of the function, the *Function* keyword is followed by the name of the function and a script block with code in it, which is delimited by curly brackets. The following example shows this pattern:

```
Function Function-Name
{
 #insert your code here
}
```

In the following Get-OperatingSystemVersion.ps1 script, the *Get-OperatingSystemVersion* function is at the top of the script. It uses the *Function* keyword to define the function followed by the name *Get-OperatingSystemVersion*.

```
Get-OperatingSystemVersion.ps1

Function Get-OperatingSystemVersion
{
 (Get-WmiObject -Class Win32_OperatingSystem).Version
} #end Get-OperatingSystemVersion

"This OS is version $(Get-OperatingSystemVersion)"
```

As shown in the preceding example, the curly brackets are open, followed by the code. The code uses the *Get-WmiObject* cmdlet to retrieve an instance of the *Win32_OperatingSystem* WMI class. Because this WMI class returns only a single instance, the properties of the class are directly accessible. The version is the property in question, and so parentheses force the evaluation of the code inside. The returned management object is used to emit the *Version* value. The curly brackets are used to close the function. The operating system version is returned to the code that calls the function. In this example, a string that writes "This OS is Version " is used. A *sub* expression is used to force evaluation of the function. The version of the operating system is returned to the place from where the function was called.

In the earlier listing of cmdlet verbs, there is one cmdlet that uses the verb *Read*. It is the *Read-Host* cmdlet, which is used to obtain information from the command line. This indicates the verb *Read* is not used to describe reading a file. There is no verb called *Display*, and the *Write* verb is used in cmdlet names such as *Write-Error* and *Write-Debug,* both of which do not really seem to represent the concept of displaying information. If you were writing a function that would read the content of a text file and display statistics about that file, you might call the function *Get-TextStatistics*. This is in keeping with cmdlet names such as *Get-Process* and *Get-Service,* which represent the concept of emitting their retrieved content within their essential functionality. The *Get-TextStatistics* function accepts a single parameter called Path. The interesting thing about parameters for functions is that when you pass a value to the parameter you use a dash, which is used to tell Windows PowerShell to use the information following the parameter to modify the way the cmdlet operates. When you refer to the value inside the function, it is a variable such as *$path*.

To call the *Get-TextStatistics* function, you have a couple of options. The first is to use the name of the function and put the value in parentheses, as shown in the following example:

```
Get-TextStatistics("C:\fso\mytext.txt")
```

This is a natural way to call the function, and it works when there is a single parameter. It does not work when there are two or more parameters. Another way to pass a value to the function is to use the dash and the parameter name, as shown in the following example:

```
Get-TextStatistics -path "C:\fso\mytext.txt"
```

You will note from the preceding example that no parentheses are required. You can also use positional arguments when passing a value. In this usage, you omit the name of the parameter entirely and simply place the value for the parameter following the call to the function, as shown in the following example:

```
Get-TextStatistics "C:\fso\mytext.txt"
```

NOTE **The use of positional arguments works well when you are working from the command line and you want to speed things along by reducing the typing load. It can be a bit confusing, and in general I tend to avoid it, even when working at the command line. This is because I often copy my working code from the console directly into a script, and as a result would need to re-type the command a second time to get rid of aliases and unnamed arguments. With the improvements in tab expansion, I feel the time saved by using positional arguments or partial arguments does not sufficiently warrant the time involved in re-typing commands when they need to be transferred to scripts. The other reason for always using named arguments is that it helps me to be aware of the exact command syntax.**

One additional way to pass a value to a function is to use partial parameter names. All that is required is enough of the parameter name to disambiguate it from other parameters. This means if you have two parameters that both begin with the letter *p*, you would need to supply enough letters of the parameter name that would separate it from the other parameter, as shown in the following example:

```
Get-TextStatistics -p "C:\fso\mytext.txt"
```

The following example shows the complete text of the *Get-TextStatistics* function:

```
Get-TextStatistics function

Function Get-TextStatistics($path)
{
 Get-Content -path $path |
 Measure-Object -line -character -word
}
```

Between Windows PowerShell 1.0 and Windows PowerShell 2.0, the number of verbs grew from 40 to 60. It is anticipated that the list of standard verbs should cover 80 to 85 percent of administrative tasks. The following list shows the Windows PowerShell 2.0 verbs:

- *Checkpoint*
- *Complete*
- *Connect*
- *Debug*
- *Disable*
- *Disconnect*
- *Enable*
- *Enter*
- *Exit*
- *Limit*
- *Receive*
- *Register*

- *Reset*
- *Restore*
- *Send*
- *Show*
- *Undo*
- *Unregister*
- *Use*
- *Wait*

The really good news is that Windows PowerShell 3.0 adds only two additional verbs:

- *Optimize*
- *Resize*

Once you have named the function, you need to create any parameters the function may require. The parameters are contained within smooth parentheses. In the *Get-TextStatistics* function, the function accepts a single parameter, the Path parameter. When you have a function that accepts a single parameter, you can pass the value to the function by placing the value for the parameter inside smooth parentheses:

```
Get-TextLength("C:\fso\test.txt")
```

The path "C:\fso\test.txt" is passed to the *Get-TextStatistics* function by means of the Path parameter. Inside the function, the string "C:\fso\text.txt" is contained in the *$path* variable. The *$path* variable lives only within the confines of the *Get-TextStatistics* function. It is not available outside the scope of the function. It is available from within child scopes of the *Get-TextStatistics* function. A child scope of the *Get-TextStatistics* is one that is created from within the *Get-TextStatistics* function.

In the Get-TextStatisticsCallChildFunction.ps1 script, the *Write-Path* function is called from within the *Get-TextStatistics* function. This means the *Write-Path* function will have access to variables that are created within the *Get-TextStatistics* function. This is the concept of *variable scope*, which is an extremely important concept when working with functions. As you use functions to separate the creation of objects, you must always be aware of where the object gets created and where you intend to use that object. In the *Get-TextStatisticsCallChildFunction,* the *$path* variable does not obtain its value until it is passed to the function. It therefore lives within the *Get-TextStatistics* function. But because the *Write-Path* function is called from within the *Get-TextStatistics* function, it inherits the variables from that scope. When you call a function from within another function, variables created within the parent function are available to the child function, as shown in the Get-TextStatisticsCallChildFunction.ps1 script:

```
Get-TextStatisticsCallChildFunction.ps1

Function Get-TextStatistics($path)
{
 Get-Content -path $path |
 Measure-Object -line -character -word
```

```
 Write-Path
}

Function Write-Path()
{
 "Inside Write-Path the `$path variable is equal to $path"
}

Get-TextStatistics("C:\fso\test.txt")
"Outside the Get-TextStatistics function `$path is equal to $path"
```

Inside the *Get-TextStatistics* function, the *$path* variable is used to provide the path to the *Get-Content* cmdlet. When the *Write-Path* function is called, nothing is passed to it. But inside the *Write-Path* function, the value of *$path* is maintained. Outside both functions, however, *$path* does not have any value. The following example shows the output from running the script:

```
               Lines               Words          Characters Property
               -----               -----          ---------- --------
                   3                  41                 210
Inside Write-Path the $path variable is equal to C:\fso\test.txt
Outside the Get-TextStatistics function $path is equal to
```

You will then need to open and close a script block. The curly bracket is used to delimit the script block on a function. As a best practice, when I write a function I always use the *Function* keyword. I type in the name, the input parameters, and the curly brackets for the script block at the same time, as shown in the following example:

```
Function My-Function()
{
 #insert your code here
}
```

In this manner, I do not forget to close the curly brackets. Trying to identify a missing curly bracket within a long script can be somewhat problematic as the error that is presented does not always correspond to the line that is missing the curly bracket. Suppose the closing curly bracket is left off the *Get-TextStatistics* function, as seen in the Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1 script. The following error will be generated:

```
Missing closing '}' in statement block.
At C:\Scripts\Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.
ps1:28 char:1
```

The problem is the position indicator of the error message points to the first character on line 28. Line 28 happens to be the first blank line after the end of the script. This means that Windows PowerShell scanned the entire script looking for the closing curly bracket. Because it did not find it, it states the error is the end of the script. If you were to place a closing curly bracket on line 28, the error in this example would disappear, but the script would not work either.

The following example shows the Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1 script with a comment that indicates where the missing closing curly bracket should be placed:

```
Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1
Function Get-TextStatistics($path)
{
 Get-Content -path $path |
 Measure-Object -line -character -word
 Write-Path
# Here is where the missing bracket goes

Function Write-Path()
{
 "Inside Write-Path the `$path variable is equal to $path"
}
Get-TextStatistics("C:\fso\test.txt")
Write-Host "Outside the Get-TextStatistics function `$path is equal to $path"
```

One other technique to guard against the missing curly bracket problem is to add a comment to the closing curly bracket of each function.

Using a type constraint

When accepting parameters for a function, it might be important to use a type constraint to ensure the function receives the correct type of data. To do this, place the desired type alias inside square brackets in front of the input parameter. This will constrain the data type and prevent the entry of an incorrect type of data. Table 12-1 shows allowable type shortcuts.

TABLE 12-1 Data type aliases

Alias	Type
[int]	32-bit signed integer
[long]	64-bit signed integer
[string]	Fixed-length string of Unicode characters
[char]	Unicode 16-bit character
[bool]	*true* or *false* value
[byte]	8-bit unsigned integer
[double]	Double-precision, 64-bit floating point number
[decimal]	A 128-bit decimal value
[single]	Single precision, 32-bit floating point number
[array]	Array of values
[xml]	XML objects
[hashtable]	A *hashtable* object (similar to a *dictionary* object)

In the *Resolve-ZipCode* function, which is included in the Resolve-ZipCode.ps1 script, the *$zip* input parameter is constrained to allow only a 32-bit signed integer for input. (Obviously the *[int]* type constraint would eliminate most of the world's zip codes, but the web service the script uses resolves only U.S. zip codes, so it is a good addition to the function.)

In the *Resolve-ZipCode* function, the first thing that is done is to use a string that points to the WSDL for the web service. Next, the *New-WebServiceProxy* cmdlet is used to create a new web service proxy for the *ZipCode* service. The WSDL for the *ZipCode* service defines a method called the *GetInfoByZip* method. It will accept a standard U.S. zip code. The results are displayed as a table. The following example shows the Resolve-ZipCode.ps1 script:

```
Resolve-ZipCode.ps1
#Requires -Version 2.0
Function Resolve-ZipCode([int]$zip)
{
 $URI = "http://www.webservicex.net/uszip.asmx?WSDL"
 $zipProxy = New-WebServiceProxy -uri $URI -namespace WebServiceProxy -class ZipClass
 $zipProxy.getinfobyzip($zip).table
} #end Get-ZipCode

Resolve-ZipCode 28273
```

When you use a type constraint on an input parameter, any deviation from the expected data type will generate an error similar to the one in the following example:

```
Resolve-ZipCode : Cannot process argument transformation on parameter 'zip'. Cannot
convert value "COW" to type "System
.Int32". Error: "Input string was not in a correct format."
At C:\Users\edwils.NORTHAMERICA\AppData\Local\Temp\tmp3351.tmp.ps1:22 char:16
+ Resolve-ZipCode <<<<  "COW"
    + CategoryInfo          : InvalidData: (:) [Resolve-ZipCode], ParameterBindin...
mationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,Resolve-ZipCode
```

Needless to say, such an error could be distracting to the users of the function. One way to handle the problem of confusing error messages is to use the Trap keyword. In the DemoTrapSystemException.ps1 script, the *My-Test* function uses *[int]* to constrain the *$myinput* variable to accepting only a 32-bit unsigned integer for input. If such an integer is received by the function when it is called, the function will return the string "It worked". If the function receives a string for input, an error will be raised, similar to the one in the preceding example.

Rather than displaying a raw error message, which most users and many IT Pros find confusing, it is a best practice to suppress the display of the error message. You also could inform the user an error condition has occurred and provide more meaningful and direct information that the user can then relay to technical support. Often, IT departments will display such an error message, complete with either a local telephone number for the appropriate technical support or even a link to an internal webpage that provides detailed troubleshooting and self-help, corrective steps the user can perform. You could even provide a webpage that hosts a script the user could run that would fix the problem. This is similar to the Microsoft Fix it Center at *http://fixitcenter.support.microsoft.com/Portal*.

When an instance of a *System.SystemException* class is created (when a system exception occurs), the *Trap* statement will trap the error rather than allowing it to display the error information on the screen. If you were to query the *$error* variable, you would see that the error had in fact occurred and was actually received by the error record. You would also have access to the *ErrorRecord* class by means of the *$_ automatic* variable, which means the error record has been passed along the pipeline. This gives you the ability to build a rich error-handling solution.

In the following example, the string "error trapped" is displayed, and the Continue statement is used to continue the script execution on the next line of code. The next line of code that is executed is the "After the error" string. The example shows the output after the DemoTrapSystemException.ps1 script is run:

```
error trapped
After the error
```

The following example shows the complete DemoTrapSystemException.ps1 script:

```
DemoTrapSystemException.ps1
Function My-Test([int]$myinput)
{

 "It worked"
} #End my-test function
# *** Entry Point to Script ***

Trap [SystemException] { "error trapped" ; continue }
My-Test -myinput "string"
"After the error"
```

Using multiple input parameters

When I use multiple input parameters, I consider it a best practice to modify the way the function is structured. This is more of a visual change that makes the function easier to read. In the basic function pattern shown in the following example, the function accepts three input parameters. When you are considering the default values and the type constraints, the parameters begin to string along fairly long. Moving them to the inside of the function body highlights the fact they are input parameters, and it makes them easier to read, easier to understand, and easier to maintain:

```
Function Function-Name
{
  Param(
        [int]$Parameter1,
        [String]$Parameter2 = "DefaultValue",
        $Parameter3
       )
#Function code goes here
} #end Function-Name
```

An example of a function that uses three input parameters is the *Get-DirectoryListing* function. With the type constraints, default values, and parameter names, the function signature would be rather cumbersome to include on a single line, as shown in the following example:

```
Function Get-DirectoryListing (String]$Path,[String]$Extension = "txt",[Switch]$Today)
```

If you increase the number of parameters to four, or if you include a default value for the *Path* parameter, the signature would easily scroll to two lines. Using the *Param* statement inside the function body also provides you with the ability to specify input parameters to a function.

> ***NOTE*** **Using the *Param* statement inside the function body is often regarded as a personal preference. Personally, I do not think it makes sense to use the *Param* statement inside the function body when there is only one or two input parameters. It requires additional work and often leaves the reader of the script wondering why this was done. When there are more than two parameters, the *Param* statement stands out visually, and it is obvious why it was done in this particular manner.**

Following the *Function* keyword and the name of the function, the *Param* keyword is used to identify the parameters for the function. Each parameter must be separated by a comma. All the parameters must be surrounded with a set of smooth parentheses. If you want to assign a default value for a parameter, such as the extension .txt for the *Extension* parameter in the *Get-DirectoryListing* function, you do a straight value assignment followed by a comma.

In the *Get-DirectoryListing* function, the *Today* parameter is a switched parameter. When it is supplied to the function, only files written to since midnight on the day the script is run will be displayed. If it is not supplied, all files matching the extension in the folder will be displayed. The following example shows the Get-DirectoryListingToday.ps1 script:

```
Get-DirectoryListingToday.ps1
Function Get-DirectoryListing
{
 Param(
       [String]$Path,
       [String]$Extension = "txt",
       [Switch]$Today
      )
 If($Today)
   {
    Get-ChildItem -Path $path\* -include *.$Extension |
    Where-Object { $_.LastWriteTime -ge (Get-Date).Date }
   }
 ELSE
  {
   Get-ChildItem -Path $path\* -include *.$Extension
  }
} #end Get-DirectoryListing

# *** Entry to script ***
Get-DirectoryListing -p c:\fso -t
```

NOTE **As a best practice, you should avoid creating functions that have a large number of input parameters. It is very confusing. When you find yourself creating a large number of input parameters, you should ask if there is a better way to accomplish your tasks. It might be an indicator you do not have a single-purpose function. In the *Get-Directory-Listing* function, I have a switched parameter that will filter the files returned by the ones written to today. If I were really writing the script for production use, instead of just to demonstrate multiple function parameters, I would have created another function called something like *Get-FilesByDate*. In that function, I would have a *Today* switch and a *Date* parameter to allow a selectable date for the filter. This separates the data-gathering function and the filter and presentation function. See the "Using functions to provide ease of modification" section later in this chapter for more discussion of this technique.**

Using functions to encapsulate business logic

Scriptwriters need to be concerned with two kinds of logic: program logic and business logic. Program logic is the way the script works, the order in which things need to be done, and the requirements of code used in the script. An example of program logic is the requirement to open a connection to a database before querying the database. Business logic is something that is a requirement of the business, but not necessarily a requirement of the program and script. The script can often operate just fine, regardless of the particulars of the business rule. If the script is designed properly, it should operate perfectly fine no matter what gets supplied for the business rules.

In the BusinessLogicDemo.ps1 script, a function called *Get-Discount* is used to calculate the discount to be granted to the total amount. One benefit to encapsulating the business rules for the discount into a function is as long as the contract between the function and the calling code does not change, you can drop any kind of convoluted discount schedule between the curly brackets of the *Get-Discount* function that the business decides to come up with. This includes database calls to determine on-hand inventory, time of day, day of week, total sales volume for the month, the buyers' loyalty level, and the square root of some random number that is used to determine the instant discount rate.

So what is the contract with the function? The contract with the *Get-Discount* function says, "If you give me a rate number as a type of *system.double* and a total as an integer, I will return to you a number that represents the total discount to be applied to the sale." As long as you adhere to that contract, you never need to modify the code.

The *Get-Discount* function begins with the Function keyword, the name of the function, and the definition for two input parameters. The first input parameter is the *$rate* parameter, which is constrained to be a *system.double* (which will permit us to supply decimal numbers). The second input parameter is the *$total* parameter, which is constrained to be a *system.integer* and therefore will not allow decimal numbers. In the script block, the value of the *-Total*

parameter is multiplied by the value of the *-Rate* parameter. The result of this calculation is returned to the pipeline.

The following example shows the *Get-Discount* function:

```
Function Get-Discount([double]$rate,[int]$total)
{
  $rate * $total
} #end Get-Discount
```

The entry point to the script assigns values to both the *$total* and *$rate* variables:

```
$rate = .05
$total = 100
```

The variable *$discount* is used to hold the result of the calculation from the *Get-Discount* function. When calling the function, it is a best practice to use full parameter names. It makes the code easier to read and will help to make it immune to unintended problems if the function signature ever changes, as shown in the following example:

```
$discount = Get-Discount -rate $rate -total $total
```

> ***NOTE*** **The signature of a function is the order and names of the input parameters. If you typically supply values to the signature by means of positional parameters, and the order of the input parameters changes, the code will fail or, worse yet, produce inconsistent results. If you typically call functions by means of partial parameter names and an additional parameter is added, the script will fail due to difficulty with the disambiguation process. Obviously, you should take this into account when first writing the script and the function, but the problem can arise months or years later when making modifications to the script or calling the function by means of another script.**

The remainder of the script produces output for the screen. The following example shows the results of running the script:

```
Total: 100
Discount: 5
Your Total: 95
```

The following example shows the complete text of the BusinessLogicDemo.ps1 script:

```
BusinessLogicDemo.ps1
Function Get-Discount([double]$rate,[int]$total)
{
 $rate * $total
} #end Get-Discount


$rate = .05
$total = 100
$discount = Get-Discount -rate $rate -total $total
```

```
"Total: $total"
"Discount: $discount"
"Your Total: $($total-$discount)"
```

Business logic does not have to be related to business purposes. Business logic is anything that is arbitrary that does not affect the running of the code. In the FindLargeDocs.ps1 script, there are two functions. The first function, *Get-Doc,* is used to find document files (files with an extension of .doc, .docx, or .dot) in a folder that is passed to the function when it is called. Use the *Recurse* switch with the *Get-ChildItem* cmdlet to cause the *Get-Doc* to look in the present folder as well as look within child folders. This function is standalone and has no dependency on any other functions.

The *LargeFiles* piece of code is a filter. A filter is kind of special purpose function that uses the Filter keyword rather than using the Function keyword when it is created:

```
FindLargeDocs.ps1
Function Get-Doc($path)
{
 Get-ChildItem -Path $path -include *.doc,*.docx,*.dot -recurse
} #end Get-Doc

Filter LargeFiles($size)
{
  $_ |
  Where-Object { $_.length -ge $size }
} #end LargeFiles

Get-Doc("C:\FSO") |  LargeFiles 1000
```

Using functions to provide ease of modification

It is a truism that a script is never completed. There is always something else to add to a script, such as a change that will improve it or additional functionality someone requests. When a script is written as one long piece of inline code, without recourse to functions, it can be rather tedious and error prone to modify.

An example of an inline script is the InLineGetIPDemo.ps1 script. The first line of code uses the *Get-WmiObject* cmdlet to retrieve the instances of the *Win32_NetworkAdapterConfiguration* WMI class that is IP-enabled. The results of this WMI query are stored in the *$IP* variable, as shown in the following example:

```
$IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
```

Once the WMI information has been obtained and stored, the remainder of the script prints information to the screen. The *IPAddress*, *IPSubNet*, and *DNSServerSearchOrder* are all stored an array. For this example, we are interested only in the first IP address, and we therefore print element *0*, which will always exist if the network adapter has an IP address. The following example shows this section of the script:

```
"IP Address: " + $IP.IPAddress[0]
"Subnet: " + $IP.IPSubNet[0]
"GateWay: " + $IP.DefaultIPGateway
"DNS Server: " + $IP.DNSServerSearchOrder[0]
"FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
```

When the script is run, it produces output similar to the following example:

```
IP Address: 192.168.2.5
Subnet: 255.255.255.0
GateWay: 192.168.2.1
DNS Server: 192.168.2.1
FQDN: w8client1.nwtraders.com
```

The following example shows the complete InLineGetIPDemo.ps1 script:

```
InLineGetIPDemo.ps1
$IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
"IP Address: " + $IP.IPAddress[0]
"Subnet: " + $IP.IPSubNet[0]
"GateWay: " + $IP.DefaultIPGateway
"DNS Server: " + $IP.DNSServerSearchOrder[0]
"FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
```

With just a few modifications to the script, you can obtain a great deal of flexibility. The modifications, of course, involve moving the inline code into functions. As a best practice, a function should be narrowly defined and should encapsulate a single thought. Although we could move the entire preceding script into a function, we would not have as much flexibility. There are two thoughts or ideas that are expressed in the script: The first is obtaining the IP information from WMI, and the second is formatting and displaying the IP information. It would be best to separate the gathering and displaying processes from one another as they are logically two different activities.

To convert the InLineGetIPDemo.ps1 script into a script that uses a function, you need to add only the *Function* keyword, give it a name, and surround the original code with a pair of curly brackets. The transformed script is now named GetIPDemoSingleFunction.ps1:

```
GetIPDemoSingleFunction.ps1
Function Get-IPDemo
{
 $IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled =
$true"
 "IP Address: " + $IP.IPAddress[0]
 "Subnet: " + $IP.IPSubNet[0]
 "GateWay: " + $IP.DefaultIPGateway
 "DNS Server: " + $IP.DNSServerSearchOrder[0]
 "FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Get-IPDemo

# *** Entry Point To Script ***

Get-IPDemo
```

If you go to all the trouble to transform the inline code into a function, what benefit do you derive? By making this single change, you gain the following benefits:

- Easier to read
- Easier to understand
- Easier to reuse
- Easier to troubleshoot

The script is easier to read because you do not really need to read each line of code to see what it does. There is a function that obtains the IP address, and it is called from outside the function. That is all the script does.

The script is easier to understand because there is a function that obtains the IP address. If you want to know the details of that operation, you can read that function. If you are not interested in the details, you can skip that portion of the code.

The script is easier to reuse because you can dot-source the script as seen here. When the script is dot-sourced, all the executable code in the script is run. As a result, because each of the scripts prints information, the following output is displayed:

```
IP Address: 192.168.2.5
Subnet: 255.255.255.0
GateWay: 192.168.2.1
DNS Server: 192.168.2.1
FQDN: w8client1.nwtraders.com


 w8client1 free disk space on drive C:
    48,767.16 MegaBytes


This OS is version 6.2
```

The following example shows the DotSourceScripts.ps1 script:

```
DotSourceScripts.ps1
. C:\Scripts\GetIPDemoSingleFunction.ps1
. C:\Scripts\Get-FreeDiskSpace.ps1
. C:\Scripts\Get-OperatingSystemVersion.ps1
```

As you can see, this provides a certain level of flexibility to choose the information required, and it also makes it easy to mix and match the required information. If each of the scripts had been written in a more standard fashion, and the output standardized, the results would have been more impressive. As it is, three lines of code produced an exceptional amount of useful output that could be acceptable in a variety of situations. The .ps1 script is easier to troubleshoot in part because it is easier to read and easier to understand.

A better way to work with the function is to think about the things the function is actually doing. In the FunctionGetIPDemo.ps1 script, there are two functions. The first connects to WMI, which returns a management object. The second function formats the output. These are two completely unrelated tasks. The first task is data gathering, and the second task is the

presentation of the information. The following example shows the FunctionGetIPDemo.ps1 script:

```
FunctionGetIPDemo.ps1
Function Get-IPObject
{
 Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
} #end Get-IPObject

Function Format-IPOutput($IP)
{
 "IP Address: " + $IP.IPAddress[0]
 "Subnet: " + $IP.IPSubNet[0]
 "GateWay: " + $IP.DefaultIPGateway
 "DNS Server: " + $IP.DNSServerSearchOrder[0]
 "FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Format-IPOutput

# *** Entry Point To Script

$ip = Get-IPObject
Format-IPOutput($ip)
```

By separating the data gathering and the presentation activities into different functions, additional flexibility is gained. You could easily modify the *Get-IPObject* function to look for network adapters that were not IP-enabled. To do this, you would need to modify the filter parameter of the *Get-WmiObject* cmdlet. Because most of the time you would actually be interested only in network adapters that are IP-enabled, it would make sense to set the default value of the input parameter to *$true*. By default, the behavior of the revised function is exactly as it was prior to modification. The advantage is you can now use the function and modify the objects returned by it. To do this, you supply *$false* when calling the function. This is illustrated in the Get-IPObjectDefaultEnabled.ps1 script:

```
Get-IPObjectDefaultEnabled.ps1
Function Get-IPObject([bool]$IPEnabled = $true)
{
 Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject

Get-IPObject -IPEnabled $False
```

By separating the gathering of the information from the presentation of the information, you gain flexibility not only in the type of information that is garnered, but also in the way the information is displayed. When gathering network adapter configuration information from a network adapter that is not enabled for IP, the results are not as impressive as one that is enabled for IP. You might therefore decide to create a different display to list only the pertinent information. As the function that displays the information is different than the one that gathers the information, you can easily make a change that customizes the information that is most germane. The *Begin* section of the function is run once during the execution of the function. This is the perfect place to create a header for the output data. The *Process* section

executes once for each item on the pipeline, which in this example will be each of the non IP-enabled network adapters. The *Write-Host* cmdlet is used to easily write the data to the Windows PowerShell console. The backtick t (*`t*) character is used to produce a tab.

> ***NOTE*** **The backtick *t* character is a string character and, as such, works with cmdlets that accept string input.**

The following example shows the Get-IPObjectDefaultEnabledFormatNonIPOutput.ps1 script:

```
Get-IPObjectDefaultEnabledFormatNonIPOutput.ps1
Function Get-IPObject([bool]$IPEnabled = $true)
{
 Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject

Function Format-NonIPOutput($IP)
{
  Begin { "Index #  Description" }
 Process {
  ForEach ($i in $ip)
  {
   Write-Host $i.Index `t $i.Description
  } #end ForEach
 } #end Process
} #end Format-NonIPOutPut

$ip = Get-IPObject -IPEnabled $False
Format-NonIPOutput($ip)
```

You can use the *Get-IPObject* function to retrieve the network adapter configuration, and you can use the *Format-NonIPOutput* and *Format-IPOutput* functions to clean up the output from the *Get-IPObject* function:

```
CombinationFormatGetIPDemo.ps1
Function Get-IPObject([bool]$IPEnabled = $true)
{
 Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject

Function Format-IPOutput($IP)
{
 "IP Address: " + $IP.IPAddress[0]
 "Subnet: " + $IP.IPSubNet[0]
 "GateWay: " + $IP.DefaultIPGateway
 "DNS Server: " + $IP.DNSServerSearchOrder[0]
 "FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Format-IPOutput

Function Format-NonIPOutput($IP)
{
  Begin { "Index #  Description" }
```

```
 Process {
  ForEach ($i in $ip)
  {
   Write-Host $i.Index `t $i.Description
  } #end ForEach
 } #end Process
} #end Format-NonIPOutPut

# *** Entry Point ***
$IPEnabled = $false
$ip = Get-IPObject -IPEnabled $IPEnabled
If($IPEnabled) { Format-IPOutput($ip) }
ELSE { Format-NonIPOutput($ip) }
```

Summary

This chapter focused on creating functions. We began with an introduction to functions and their use. Next, we discussed how to accept parameters for input and concluded by demonstrating two special case scenarios for using functions.

CHAPTER 13

Debugging scripts

- Understanding debugging in Windows PowerShell
- Debugging the script

No one enjoys debugging scripts. In fact, the best debugging is no debugging. It is also true that well written, well formatted, well documented, and clearly constructed Windows PowerShell code requires less effort to debug than poorly formatted, undocumented spaghetti code. It is fair to say that debugging begins when you first open the Windows PowerShell ISE.

Understanding debugging in Windows PowerShell

If you can read and understand your Windows PowerShell code, chances are you will need to do very little debugging. But what if you do need to do some debugging? Well, just as excellent golfers spend many hours practicing chipping out of the sand trap in hopes that they will never need to use the skill, so too must competent Windows PowerShell scripters practice debugging skills in hopes they will never need to apply the knowledge. Understanding the color coding of the Windows PowerShell ISE, detecting when closing quotation marks are missing, and knowing which pair of braces correspond to which command can greatly reduce the debugging that might be needed later.

Debugging the script

The debugging features of Windows PowerShell 3.0 make the use of the *Set-PSDebug* cmdlet seem rudimentary or even cumbersome. Once you are more familiar with the debugging features of Windows PowerShell 3.0, you might decide not to use the *Set-PSDebug* cmdlet. Several cmdlets enable debugging from both the Windows PowerShell console and from the Windows PowerShell ISE. Table 13-1 shows the debugging cmdlets.

TABLE 13-1 Windows PowerShell debugging cmdlets

Cmdlet name	Cmdlet function
Set-PsBreakpoint	Sets breakpoints on lines, variables, and commands.
Get-PsBreakpoint	Gets breakpoints in the current session.
Disable-PsBreakpoint	Turns off breakpoints in the current session.
Enable-PsBreakpoint	Re-enables breakpoints in the current session.
Remove-PsBreakpoint	Deletes breakpoints from the current session.
Get-PsCallStack	Displays the current call stack.

Setting breakpoints

The debugging features in Windows PowerShell use *breakpoints*. Breakpoints are very familiar to developers who have used products such as Microsoft Visual Studio in the past. But for many IT pros without a programming background, the concept of a breakpoint is somewhat foreign. A breakpoint is a spot in the script where you would like the execution of the script to pause. Because the script pauses, it is like the stepping functionality that we examined earlier. But because you control where the breakpoint will occur, the stepping experience is much faster because it doesn't halt on each line of the script. In addition, because you have many different methods to use to set the breakpoint (instead of merely stepping through the script line by line), you can tailor the breakpoint to reveal precisely the information you are looking for.

Setting a breakpoint on a line number

To set a breakpoint, use the *Set-PSBreakpoint* cmdlet. The easiest way to set a breakpoint is to set it on line 1 of the script. To set a breakpoint on the first line of the script, use the Line parameter and Script parameter. When you set a breakpoint, an instance of the System.Management.Automation.LineBreak .NET Framework class is returned. It lists the ID, Script, and Line properties that were assigned when the breakpoint was created, as shown in the following example:

```
PS C:\> Set-PSBreakpoint -line 1 -script Y:\BadScript.ps1
  ID Script            Line Command           Variable         Action
  -- ------            ---- -------           --------         ------
   0 BadScript.ps1        1
```

This causes the script to break immediately. You can then step through the function in the same way you did using the *Set-PSDebug* cmdlet with the Step parameter. When you run the script, it hits the breakpoint that was set on the first line of the script, and Windows PowerShell enters the script debugger. Windows PowerShell will enter the debugger every time the BadScript.ps1 script is run from the Y drive. When the Windows PowerShell enters the debugger, the Windows PowerShell command prompt changes to [DBG]: PS C:\>>> to visually alert

you that you are inside the Windows PowerShell debugger. To step to the next line in the script, type s. To quit the debugging session, type q. The debugging commands are not case sensitive. The following example shows this process:

```
PS C:\> Y:\BadScript.ps1
Hit Line breakpoint on 'Y:\BadScript.ps1:1'

BadScript.ps1:1   #
------------------------------------------------------------------------
[DBG]: PS C:\>>> s
BadScript.ps1:16  Function AddOne([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:21  Function AddTwo([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:26  Function SubOne([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:31  Function TimesOne([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:36  Function TimesTwo([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:41  Function DivideNum([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:48  $num = 0
[DBG]: PS C:\>>> s
BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
BadScript.ps1:28   $num-1
[DBG]: PS C:\>>> s
BadScript.ps1:43   12/$num
[DBG]: PS C:\>>> s
                                    if ($_.FullyQualifiedErrorId -ne
"NativeCommandErrorMessage" -and $ErrorView -ne "CategoryView") {
[DBG]: PS C:\>>> q
PS C:\>
```

NOTE **Keep in mind that breakpoints depend on the location of the specific script when you specify a breakpoint on a script. When you create a breakpoint for a script, you specify the location to the script for which you want to set a breakpoint. Often I have several copies of a script that I keep in different locations (for version control). At times, I get confused in a long debug session and might open up the wrong version of the script to debug it. This will not work. If the script is identical in all respects except for the path to the script, it will not break. If you want to use a single breakpoint that could apply to a specific script that is stored in multiple locations, you can set the breakpoint for the condition inside the Windows PowerShell console. In this case, do not use the *Script* parameter.**

Setting a breakpoint on a variable

Setting a breakpoint on line 1 of the script is useful for easily entering a debug session, but setting a breakpoint on a variable can often make a problem with a script easy to detect. This is, of course, especially true when you have already determined the problem is with a variable that is either getting assigned a value or being ignored. There are three modes that can be configured for when the breakpoint is specified for a variable. The modes are specified by using the Mode parameter. Table 13-2 lists the three modes of operation.

TABLE 13-2 Variable breakpoint access modes

Access Mode	Meaning
Write	Stops execution immediately before a new value is written to the variable.
Read	Stops execution when the variable is read; that is, when its value is accessed, either to be assigned, displayed, or used. In *Read* mode, execution does not stop when the value of the variable changes.
ReadWrite	Stops execution when the variable is read or written.

To see when the BadScript.ps1 script writes to the *$num* variable, use the Write mode. When you specify the value for the Variable parameter, do not include the dollar sign in front of the variable name. To set a breakpoint on a variable, you need to supply only a path to the script, the name of the variable, and the access mode. When a variable breakpoint is set, the System.Management.Automation.LineBreak .NET Framework class object that is returned does not include the access mode value. This is true even if you use the *Get-PSBreakpoint* cmdlet to directly access the breakpoint. If you pipeline the *System.Management.Automation.LineBreak* .NET Framework class object to the *Format-List* cmdlet, you will be able to see that the access mode property is available. In the following example, we set a breakpoint when the *$num* variable is written to in the y:\BadScript.ps1 script.

```
PS C:\> Set-PSBreakpoint -Variable num -Mode write -Script Y:\BadScript.ps1
  ID Script            Line Command           Variable         Action
  -- ------            ---- -------           --------         ------
   3 BadScript.ps1                            num

PS C:\> Get-PSBreakpoint
  ID Script            Line Command           Variable         Action
  -- ------            ---- -------           --------         ------
   3 BadScript.ps1                            num

PS C:\> Get-PSBreakpoint  | Format-List * -Force
AccessMode : Write
Variable   : num
Action     :
Enabled    : True
HitCount   : 0
Id         : 3
Script     : Y:\BadScript.ps1
```

After you set the breakpoint, when you run the script (if the other breakpoints have been removed or deactivated, which will be discussed later) the script will enter the Windows PowerShell debugger when the breakpoint is hit; in other words, the breakpoint is hit when the value of the *$num* variable is written to. If you step through the script by using the s command, you will be able to follow the sequence of operations. Only one breakpoint is hit when the script is run. This was on line 48 when the value was set to 0, as shown in the following example:

```
PS C:\> Y:\BadScript.ps1
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (Write access)

BadScript.ps1:48  $num = 0
[DBG]: PS C:\>>> $num
[DBG]: PS C:\>>> Write-Host $num

[DBG]: PS C:\>>> s
BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> $num
0
```

To set a breakpoint on a *Read* operation for the variable, you specify the Variable parameter and name of the variable, the Script parameter with the path to the script, and *Read* as the value for the Mode parameter, as shown in the following example:

```
PS C:\> Set-PSBreakpoint -Variable num -Script Y:\BadScript.ps1 -Mode read

  ID Script            Line Command           Variable         Action
  -- ------            ---- -------           --------         ------
   4 BadScript.ps1                            num
```

When you run the script, a breakpoint will be displayed each time you hit a *Read* operation on the variable. Each breakpoint will be displayed in the Windows PowerShell console as `Hit Variable breakpoint` followed by the path to the script and the access mode of the variable. In the BadScript.ps1 script, the value of the *$num* variable is read several times. The following example shows the truncated output:

```
PS C:\> Y:\BadScript.ps1
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (Read access)

BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (Read access)

BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
BadScript.ps1:28   $num-1
[DBG]: PS C:\>>> s
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (Read access)

BadScript.ps1:28   $num-1
[DBG]: PS C:\>>> s
```

If you set the ReadWrite access mode for the Mode parameter for the variable *$num* for the BadScript.ps1 script, you will receive the following output:

```
PS C:\> Set-PSBreakpoint -Variable num -Mode readwrite -Script Y:\BadScript.ps1

  ID Script            Line Command           Variable         Action
  -- ------            ---- -------           --------         ------
   6 BadScript.ps1                            num
```

When you run the script (assuming you have disabled the other breakpoints), you will hit a breakpoint each time the *$num* is read to or written to. If you get tired of typing **s** and pressing Enter while you are in the debugging session, you can press Enter and it will repeat your previous s command as you continue to step through the breakpoints. When the script has stepped through the code and hits the error in the BadScript.ps1 script, type to exit the debugger. The following example shows this process:

```
PS C:\> Y:\BadScript.ps1
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:48  $num = 0
[DBG]: PS C:\>>> s
BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>>
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>>
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>>
BadScript.ps1:28   $num-1
[DBG]: PS C:\>>>
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:28   $num-1
[DBG]: PS C:\>>>
BadScript.ps1:43   12/$num
[DBG]: PS C:\>>>
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:43   12/$num
[DBG]: PS C:\>>>
                                  if ($_.FullyQualifiedErrorId -ne
"NativeCommandErrorMessage" -and $ErrorView -ne "CategoryView") {
[DBG]: PS C:\>>> q
PS C:\>
```

When you use the ReadWrite access mode of the Mode parameter for breaking on variables, the breakpoint does not tell you if the operation was a *Read* operation or a *Write* operation. You have to look at the code that is being executed to determine if the value of the variable was being written or read.

By specifying a value for the Action parameter, you can include regular Windows PowerShell code that will execute when the breakpoint is hit. If, for example, you are trying to follow the value of a variable within the script, and you want to display the value of the variable each time the breakpoint is hit, you might want to specify an Action that uses the *Write-Host* cmdlet to display the value of the variable. By using the *Write-Host* cmdlet, you can also include a string that indicates the value of the variable being displayed. This is crucial for picking up variables that never initialize; therefore, it is easier to spot than a blank line. The following example shows the technique of using the *Write-Host* cmdlet in an Action parameter:

```
PS C:\> Set-PSBreakpoint -Variable num -Action { write-host "num = $num" ;
Break } -Mode readwrite -script Y:\BadScript.ps1

  ID Script            Line Command           Variable         Action
  -- ------            ---- -------           --------         ------
   5 BadScript.ps1                            num               write-host "...
```

When you run the Y:\BadScript.ps1 with the breakpoint set, you receive the following output inside the Windows PowerShell debugger:

```
PS C:\> Y:\BadScript.ps1
num =
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:48  $num = 0
[DBG]: PS C:\>>> s
BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
Set-PSBreakpoint -Variable num -Action { write-host "num = $num" ; break }
-Mode readwrite -script Y:\BadScript.ps1
[DBG]: PS C:\>>> s
num = 0
Set-PSBreakpoint -Variable num -Action { write-host "num = $num" ; break }
-Mode readwrite -script Y:\BadScript.ps1
[DBG]: PS C:\>>> c
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>>
```

Setting a breakpoint on a command

To set the breakpoint on a command, use the Command parameter. You can break on a call to a Windows PowerShell cmdlet, function, or external script. You can use aliases when setting breakpoints. When you create a breakpoint on an alias for a cmdlet, the debugger will hit only on the use of the alias, not on the actual command name. In addition, you do not have to specify a script for the debugger to break. If you do not type a path to a script, the debugger will be active for everything within the Windows PowerShell console session. Every occurrence of the ForEach command causes the debugger to break. Because ForEach is a language statement as well as an alias for the *ForEach-Object* cmdlet, you might wonder whether the Windows PowerShell debugger will break on both the language statement and the use of the

alias for the cmdlet. The answer is no. You can set breakpoints on language statements, but the debugger will not break on a language statement. As shown in the following example, the debugger breaks on the use of the *ForEach* alias, but not on the use of the *ForEach-Object* cmdlet:

```
PS C:\> Set-PSBreakpoint -Command foreach

  ID Script            Line Command          Variable         Action
  -- ------            ---- -------          --------         ------
  10                        foreach


PS C:\> 1..3 | ForEach-Object { $_}
1
2
3
PS C:\> 1..3 | foreach { $_ }
Hit Command breakpoint on 'foreach'

1..3 | foreach { $_ }
[DBG]: PS C:\>>> c
1
Hit Command breakpoint on 'foreach'

1..3 | foreach { $_ }
[DBG]: PS C:\>>> c
2
Hit Command breakpoint on 'foreach'

1..3 | foreach { $_ }
[DBG]: PS C:\>>> c
3
```

> ***NOTE*** **You can use the shortcut technique of creating the breakpoint for the Windows PowerShell session, and not specifically for the script. By leaving out the Script parameter when creating a breakpoint, you cause the debugger to break into any running script that uses the named function. This allows you to use the same breakpoints when debugging scripts that use the same function.**

When you create a breakpoint for the *DivideNum* function used by the y:\BadScript.ps1 script, you can omit the path to the script because it is the only script that uses the *Divide-Num* function. This makes the command easier to type, but could become confusing when you look through a collection of breakpoints. If you are debugging multiple scripts in a single Windows PowerShell console session, it could become confusing if you do not specify the script to which the breakpoint applies unless you are specifically debugging the function as it

is used in multiple scripts. The following example shows the creation of a command breakpoint for the *DivideNum* function:

```
PS C:\> Set-PSBreakpoint -Command DivideNum

  ID Script            Line Command           Variable         Action
  -- ------            ---- -------           --------         ------
   7                       DivideNum
```

When you run the script, it hits a breakpoint when the *DivideNum* function is called. When BadScript.ps1 hits the *DivideNum* function, the value of *$num* is *0*. As you step through the *DivideNum* function, you assign the value of *2* to the *$num* variable. The result of *6* is displayed, then the *12/$num* operation is carried out. Next, the *AddOne* function is called, and the value of *$num* is once again *0*. When the *AddTwo* function is called, the value of *$num* is also *0*. The following example shows this process:

```
PS C:\> Y:\BadScript.ps1
Hit Command breakpoint on 'DivideNum'

BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
BadScript.ps1:43   12/$num
[DBG]: PS C:\>>> $num
0
[DBG]: PS C:\>>> $num =2
[DBG]: PS C:\>>> s
6
BadScript.ps1:50  AddOne($num) | AddTwo($num)
[DBG]: PS C:\>>> s
BadScript.ps1:18   $num+1
[DBG]: PS C:\>>> $num
0
[DBG]: PS C:\>>> s
BadScript.ps1:23   $num+2
[DBG]: PS C:\>>> $num
0
[DBG]: PS C:\>>> s
2
PS C:\>
```

Responding to breakpoints

When the script reaches a breakpoint, control of the Windows PowerShell console is turned over to you. Inside the debugger, you can type any legal Windows PowerShell command and even run cmdlets such as *Get-Process* or *Get-Service*. In addition, there are several new debugging commands you can type into the Windows PowerShell console when a breakpoint has been reached. Table 13-3 shows the available debug commands.

TABLE 13-3 Windows PowerShell Debugger commands

Keyboard shortcut	Command name	Command meaning
s	*Step-into*	Executes the next statement and then stops.
v	*Step-over*	Executes the next statement, but skips functions and invocations. The skipped statements are executed, but not stepped through.
o	*Step-out*	Steps out of the current function up one level if nested. If in the main body, it continues to the end or the next breakpoint. The skipped statements are executed, but not stepped through.
c	*Continue*	Continues to run until the script is complete or until the next breakpoint is reached. The skipped statements are executed, but not stepped through.
l	*List*	Displays the part of the script that is executing. By default, it displays the current line, 5 previous lines, and 10 subsequent lines. To continue listing the script, press Enter.
l <m>	*List*	Displays 16 lines of the script beginning with the line number specified by <m>.
l <m> <n>	*List*	Displays <n> lines of the script, beginning with the line number specified by <m>.
q	*Stop*	Stops executing the script, and exits the debugger.
k	*Get-PsCallStack*	Displays the current call stack.
Enter	*Repeat*	Repeats the last command if it was *Step* (*s*), *Step-over* (*v*), or *List* (*l*). Otherwise, represents a submit action.
h or ?	*Help*	Displays the debugger command-line Help.

Using the *DivideNum* function as a breakpoint, when the BadScript.ps1 script is run the script breaks on line 49 when the *DivideNum* function is called. The s debugging command is used to step into the next statement and stop prior to actually executing the command. The l debugging command is used to list the 5 previous lines of code from the BadScript.ps1 script and the 10 lines of code that follow the current line in the script, as shown in the following example:

```
PS C:\> Y:\BadScript.ps1
Hit Command breakpoint on 'Y:\BadScript.ps1:dividenum'

BadScript.ps1:49  SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
BadScript.ps1:43   12/$num
[DBG]: PS C:\>>> l

   38:   $num*2
   39:  } #end function TimesTwo
   40:
   41:  Function DivideNum([int]$num)
   42:  {
   43:*  12/$num
```

```
   44:  } #end function DivideNum
   45:
   46:  # *** Entry Point to Script ***
   47:
   48:  $num = 0
   49:  SubOne($num) | DivideNum($num)
   50:  AddOne($num) | AddTwo($num)
   51:
```

After reviewing the code, the o debugging command is used to step out of the *DivideNum* function. The remaining code in the *DivideNum* function is still executed, and therefore the Divide By Zero error is displayed. There are no more prompts until the next line of executing code is met. The v debugging statement is used to step over the remaining functions in the script. The remaining functions are still executed, and the results are displayed at the Windows PowerShell console:

```
[DBG]: PS C:\>>> o
Attempted to divide by zero.
At Y:\BadScript.ps1:43 char:5
+  12/ <<<< $num
    + CategoryInfo          : NotSpecified: (:) [], RuntimeException
    + FullyQualifiedErrorId : RuntimeException

BadScript.ps1:50  AddOne($num) | AddTwo($num)
[DBG]: PS C:\>>> v
2
PS C:\>
```

Listing breakpoints

Once you have set several breakpoints, you might want to know where they have been created. One thing to keep in mind is that breakpoints are stored in the Windows PowerShell environment and not in the individual script. Using the debugging features does not involve editing of the script or modifying your source code. This enables you to debug any script without worry of corrupting the code. But because you might have set several breakpoints in the Windows PowerShell environment during a typical debugging session, you might want to know what breakpoints have been defined. To do this, use the *Get-PSBreakpoint* cmdlet, as shown in the following example:

```
PS C:\> Get-PSBreakpoint
  ID Script            Line Command          Variable         Action
  -- ------            ---- -------          --------         ------
  11 BadScript.ps1          dividenum
  13 BadScript.ps1          if
   3 BadScript.ps1                          num
   5 BadScript.ps1                          num
   6 BadScript.ps1                          num
   7                        DivideNum
   8                        foreach
   9                        gps
  10                        foreach
PS C:\>
```

If you are interested in which breakpoints are currently enabled, you need to use the *Where-Object* cmdlet and pipeline the results of the *Get-PSBreakpoint* cmdlet, as shown in the following example:

```
PS C:\> Get-PSBreakpoint | where { $_.enabled }

  ID Script            Line Command           Variable         Action
  -- ------            ---- -------           --------         ------
  11 BadScript.ps1          dividenum


PS C:\>
```

You could also pipeline the results of the *Get-PSBreakpoint* to a *Format-Table* cmdlet:

```
PS C:\> Get-PSBreakpoint |
Format-Table -Property id, script, command, variable, enabled -AutoSize

Id Script           Command   variable Enabled
-- ------           -------   -------- -------
11 Y:\BadScript.ps1 dividenum             True
13 Y:\BadScript.ps1 if                   False
 3 Y:\BadScript.ps1           num        False
 5 Y:\BadScript.ps1           num        False
 6 Y:\BadScript.ps1           num        False
 7                  DivideNum            False
 8                  foreach              False
 9                  gps                  False
10                  foreach              False
```

Because the creation of the custom formatted breakpoint table requires a little bit of typing, and because the display is extremely helpful, you might consider placing the code into a function that could be included in your profile or in a custom debugging module. The following example shows such a function stored in the Get-EnabledBreakpointsFunction.ps1 script:

```
Get-EnabledBreakpointsFunction.ps1

Function Get-EnabledBreakpoints
{
  Get-PSBreakpoint |
  Format-Table -Property id, script, command, variable, enabled -AutoSize
}

# *** Entry Point to Script ***

Get-EnabledBreakpoints
```

Enabling and disabling breakpoints

While you are debugging a script, you might need to disable a particular breakpoint to see how the script runs. To do this, use the *Disable-PSBreakpoint* cmdlet:

```
Disable-PSBreakpoint -id 0
```

On the other hand, you might also need to enable a breakpoint. To do this, use the *Enable-PSBreakpoint* cmdlet:

```
Enable-PSBreakpoint -id 1
```

As a best practice, when I am in a debugging session, I use the selectively enable and disable breakpoints to see how the script is running in an attempt to troubleshoot the script. To keep track of the status of breakpoints, I use the *Get-PSBreakpoint* cmdlet, as illustrated in the preceding section.

Deleting breakpoints

When you are finished debugging the script, you should remove all the breakpoints that were created during the Windows PowerShell session. There are two ways to do this. The first way is to close the Windows PowerShell console. While this is a good way to clean up the environment, you might not want to do this because you could have remote Windows PowerShell sessions defined or variables that are populated with the results of certain queries. The second way to delete all the breakpoints is to use the *Remove-PSBreakpoint* cmdlet. Unfortunately, there is no All switch for the *Remove-PSBreakpoint* cmdlet. When you delete a breakpoint, you must supply the breakpoint ID number for the *Remove-PSBreakpoint* cmdlet. To remove a single breakpoint, you specify the ID number for the id parameter, as shown in the following example:

```
Remove-PSBreakpoint -id 3
```

If you want to remove all the breakpoints, pipeline the results from *Get-PSBreakpoint* to *Remove-PSBreakpoint,* as shown in the following example:

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

If you want to remove only the breakpoints from a specific script, you can pipeline the results through the *Where* object:

```
(Get-PSBreakpoint | Where ScriptName - eq "C:\Scripts\Test.ps1") |
Removeakpoint
```

Summary

This chapter discussed using the Windows PowerShell debugger. We covered setting breakpoints, listing breakpoints, and enabling and disabling breakpoints. The chapter concluded with a discussion about deleting breakpoints.

CHAPTER 14

Handling errors

- Handling missing parameters
- Limiting choices
- Handling missing rights
- Using *Try/Catch/Finally*

When it comes to handling errors in your script, you need to have an understanding of the intended use of the script. The way that a script will be used is sometimes called the *use case scenario*. It describes how the user will interact with the script. If the use case scenario is simple, the user might not need to do anything more than type the name of the script inside the Windows PowerShell console. A script such as Get-Bios.ps1 could get by without much need for any error handling. This is because there are no inputs to the script. The script is called, it runs, and it displays information that should always be readily available because the *Win32_Bios* WMI class is present in all versions of Windows since Windows 2000:

```
Get-Bios.ps1
Get-WmiObject -class Win32_Bios
```

Handling missing parameters

When you examine the Get-Bios.ps1 script, you can see that it does not receive any input from the command line. This is a good way to avoid user errors in your script, but it is not always practical. When your script accepts command-line input, you are opening the door for all kinds of potential problems. Depending on how you accept command-line input, you might need to test the input data to ensure that it corresponds to the type of input the script is expecting. The Get-Bios.ps1 script does not accept command-line input; therefore, it avoids most potential sources of errors.

Creating a default value for the parameter

There are two ways to assign default values for a command-line parameter. You can assign the default value in the Param declaration statement or you can assign the value in the script itself. Given a choice between the two, it is a best practice to assign the default value in the Param statement. This is because it makes the script easier to read, which in turn makes the script easier to modify and easier to troubleshoot. For more information on troubleshooting scripts, see Chapter 13, "Debugging Scripts."

Detecting the missing value and assigning it in the script

In the Get-BiosInformation.ps1 script, a command-line parameter, computerName, allows the script to target both local and remote computers. If the script runs without a value for the computerName parameter, the *Get-WmiObject* cmdlet fails because it requires a value for the computerName parameter. To solve the problem of the missing parameter, the Get-BiosInformation.ps1 script checks for the presence of the *$computerName* variable. If this variable is missing, it means it was not created through the command-line parameter, and the script therefore assigns a value to the *$computerName* variable. The following example shows the line of code that populates the value of the *$computerName* variable:

```
If(-not($computerName)) { $computerName = $env:computerName }
```

The following example shows the completed get-BiosInformation.ps1 script:

```
Get-BiosInformation.ps1
Param(
  [string]$computerName
) #end param

Function Get-BiosInformation($computerName)
{
 Get-WmiObject -class Win32_Bios -computerName $computername
} #end function Get-BiosName

# *** Entry Point To Script ***
If(-not($computerName)) { $computerName = $env:computerName }
Get-BiosInformation -computerName $computername
```

Assigning the value in the *Param* statement

To assign a default value in the Param statement, use the equality operator following the parameter name and assign the value to the parameter, as shown in the following example:

```
Param(
  [string]$computerName = $env:computername
) #end param
```

The advantage of assigning the default value for the parameter in the Param statement is the script is easier to read. Because the parameter declaration and the default parameter are in the same place, you can see immediately which parameters have default values and

which ones do not have default values. The second advantage that arises from assigning a default value in the Param statement is the script is easier to write. You will notice there is no If statement used to check the existence of the *$computerName* variable. The following example shows the complete Get-BiosInformationDefaultParam.ps1 script:

```
Get-BiosInformationDefaultParam.ps1
Param(
  [string]$computerName = $env:computername
) #end param

Function Get-BiosInformation($computerName)
{
 Get-WmiObject -class Win32_Bios -computername $computername
} #end function Get-BiosName

# *** Entry Point To Script ***

Get-BiosInformation -computerName $computername
```

Making the parameter mandatory

The best way to handle an error is to ensure the error does not occur in the first place. In Windows PowerShell 3.0, you can mark a parameter mandatory. The advantage of marking a parameter mandatory is it requires the user of the script to supply a value for the parameter. If you do not want the user of the script to be able to run the script without making a particular selection, you should make the parameter mandatory. To make a parameter mandatory, use the *Mandatory* parameter attribute. The following example shows this technique:

```
Param(
   [Parameter(Mandatory=$true)]
   [string]$drive,
   [string]$computerName = $env:computerName
) #end param
```

The following example shows the complete MandatoryParameter.ps1 script:

```
MandatoryParameter.ps1
#Requires -version 3.0
Param(
   [Parameter(Mandatory=$true)]
   [string]$drive,
   [string]$computerName = $env:computerName
) #end param

Function Get-DiskInformation($computerName,$drive)
{
 Get-WmiObject -class Win32_volume -computername $computername `
-filter "DriveLetter = '$drive'"
} #end function Get-BiosName

# *** Entry Point To Script ***

 Get-DiskInformation -computername $computerName -drive $drive
```

When a script with a mandatory parameter runs without supplying a value for the parameter, an error does not generate. Instead, Windows PowerShell prompts for the required parameter value, as shown in the following example:

```
PS C:\bp> .\MandatoryParameter.ps1

cmdlet MandatoryParameter.ps1 at command pipeline position 1
Supply values for the following parameters:
drive:
```

Limiting choices

Depending on the design of the script, several scripting techniques can ease error-checking requirements. If you have a limited number of choices you want to display to your user, you can use the PromptForChoice method. If you want to limit the selection to computers that are currently running, you can use the *Test-Connection* cmdlet prior to attempting to connect to a remote computer. If you would like to limit the choice to a specific subset of computers or properties, you can parse a text file and use the contains operator. In this section, we examine each of these techniques for limiting the permissible input values from the command line.

Using PromptForChoice to limit selections

If you use the PromptForChoice method of soliciting input from the user, your user has a limited number of options from which to choose. You eliminate the problem of bad input because the user has only specific options available to supply to your script. Figure 14-1 shows the user prompt from the PromptForChoice method.

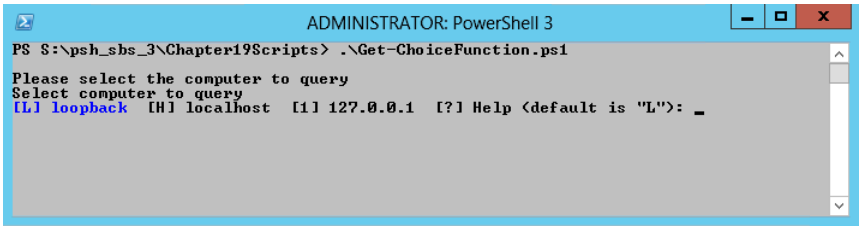


FIGURE 14-1 The *PromptForChoice* method presents a selectable menu to the user.

The use of the PromptForChoice method appears in the Get-ChoiceFunction.ps1 script. In the *Get-Choice* function, the *$caption* variable and the *$message* variable hold the caption and message that is used by PromptForChoice. The choices that are offered are an instance of the ChoiceDescription .NET Framework class. When you create the ChoiceDescription class, you also supply an array with the choices that will appear:

```
$choices = [System.Management.Automation.Host.ChoiceDescription[]] `
 @("&loopback", "local&host", "&127.0.0.1")
```

Next, you need to select a number to represent the default choice. When you begin counting, keep in mind that *ChoiceDescription* is an array and the first option is numbered *0*. Next, call the *PromptForChoice* method and display the options:

```
[int]$defaultChoice = 0
$choiceRTN = $host.ui.PromptForChoice($caption,$message, $choices,$defaultChoice)
```

Because the *PromptForChoice* method returns an integer, you could use the If statement to evaluate the value of the *$choiceRTN* variable. The syntax of the switch statement is more compact and is actually a better choice for this application. The following example shows the *Switch* statement from the *Get-Choice* function:

```
switch($choiceRTN)
 {
  0    { "loopback"  }
  1    { "localhost"  }
  2    { "127.0.0.1"  }
 }
```

When you call the *Get-Choice* function, it returns the computer that was identified by the *PromptForChoice* method. You place the method call in a set of parentheses to force it to be evaluated before the rest of the command:

```
Get-WmiObject -class win32_bios -computername (Get-Choice)
```

This solution to the problem of bad input works well when you have technical support personnel who will be working with a limited number of computers. The other caveat to this approach is that you do not want to have to change the choices on a regular basis, so you would want a stable list of computers to avoid creating a maintenance nightmare for yourself. The following example shows the complete Get-ChoiceFunction.ps1 script:

```
Get-ChoiceFunction.ps1
Function Get-Choice
{
 $caption = "Please select the computer to query"
 $message = "Select computer to query"
 $choices = [System.Management.Automation.Host.ChoiceDescription[]] `
 @("&loopback", "local&host", "&127.0.0.1")
 [int]$defaultChoice = 0
 $choiceRTN = $host.ui.PromptForChoice($caption,$message, $choices,$defaultChoice)

 switch($choiceRTN)
 {
  0    { "loopback"  }
  1    { "localhost"  }
  2    { "127.0.0.1"  }
 }
} #end Get-Choice function

Get-WmiObject -class win32_bios -computername (Get-Choice)
```

Using *Test-Connection* to identify accessible computers

If you have more than a few computers that need to be accessible, or if you do not have a stable list of computers you will be working with, then one solution to the problem of trying to connect to non-existent computers is to ping the computer prior to attempting to make the WMI connection.

You can use the *Win32_PingStatus* WMI class to send a ping to a computer. The best way to use the *Win32_PingStatus* WMI class is to use the *Test-Connection* cmdlet because it wraps the WMI class into an easy-to use-package. The following example shows how to use the *Test-Connection* cmdlet with default values:

```
PS C:\> Test-Connection -ComputerName dc1

Source        Destination     IPV4Address      IPV6Address
------        -----------     -----------      -----------
W8CLIENT6     dc1             192.168.0.101
W8CLIENT6     dc1             192.168.0.101
W8CLIENT6     dc1             192.168.0.101
W8CLIENT6     dc1             192.168.0.101
```

If you are interested only if the target computer is up or not, use the Quiet parameter. The Quiet parameter returns a Boolean value (*true* if the computer is up, *false* if the computer is down):

```
PS C:\> Test-Connection -ComputerName dc1 -Quiet
True
```

The *Test-Connection* cmdlet tends to be slower than the traditional Ping utility. It has a lot more capabilities and even returns an object, but it is slower. A few seconds can make a huge difference when attempting to run a single script to manage thousands of computers. To increase performance in these types of fan-out scenarios, use the Count parameter to reduce the default number of pings from four to one. In addition, reduce the default buffersize from 32 to 16.

Because *Test-Connection -Quiet* returns a Boolean value, it means there is no need to evaluate a number of possible return values. In fact, the logic is simple: Either it returns, or it does not. If it does return, add the action to take place in the If statement. If it does not return, add the action to take in the Else statement. Or perhaps you do not want to log failed connections. In that case, you would only have to contend with the action in the If statement. The Test-ComputerPath .psl script illustrates using the *Test-Connection* cmdlet to determine if a computer is up prior to attempting a remote connection. The following example shows the complete Test-ComputerPath script:

```
Test-ComputerPath.ps1
Param([string]$computer = $env:COMPUTERNAME)
if(Test-Connection -computer $computer -BufferSize 16 -Count 1 -Quiet)
 { Get-WmiObject -class Win32_Bios -computer $computer }
Else
 { "Unable to reach $computer computer"}
```

Using the *contains* operator to examine contents of an array

To verify input that is received from the command line, you can use the contains operator to examine the contents of an array of possible values. In the following example, an array of three values is created and stored in the variable *$noun*. The contains operator is then used to see if the array contains "hairy-nosed wombat." Because the *$noun* variable does not have an array element that is equal to the string "hairy-nosed wombat," the contains operator returns *false*:

```
PS C:\> $noun = "cat","dog","rabbit"
PS C:\> $noun -contains "hairy-nosed wombat"
False
PS C:\>
```

If an array contains a match, the contains operator returns *true*:

```
PS C:\> $noun = "cat","dog","rabbit"
PS C:\> $noun -contains "rabbit"
True
PS C:\>
```

The contains operator returns *true* only when there is an exact match. Partial matches return *false*:

```
PS C:\> $noun = "cat","dog","rabbit"
PS C:\> $noun -contains "bit"
False
PS C:\>
```

The contains operator is case insensitive. Therefore it will return *true* when matched, regardless of case:

```
PS C:\> $noun = "cat","dog","rabbit"
PS C:\> $noun -contains "Rabbit"
True
PS C:\>
```

If you need to perform a case-sensitive match, you can use the case-sensitive version of the contains operator ccontains. As shown in the following example, it will return *true* only if the case of the string matches the value contained in the array:

```
PS C:\> $noun = "cat","dog","rabbit"
PS C:\> $noun -ccontains "Rabbit"
False
PS C:\> $noun -ccontains "rabbit"
True
PS C:\>
```

In the Get-AllowedComputers.ps1 script, a single command-line parameter is created that is used to hold the name of the target computer for the WMI query. The computer parameter is a string, and it receives the default value from the environmental drive. This is a good technique because it ensures the script will have the name of the local computer, which could then be used in producing a report of the results. If you set the value of the computer parameter to *localhost*, you will never know what computer the results belong to, as shown in the following example:

```
Param([string]$computer = $env:computername)
```

The *Get-AllowedComputer* function is used to create an array of permitted computer names and to check the value of the *$computer* variable to see if it is present. If the value of the *$computer* variable is present in the array, the *Get-AllowedComputer* function returns *true*. If the value is missing from the array, the *Get-AllowedComputer* function returns *false*. The array of computer names is created by using the *Get-Content* cmdlet to read a text file that contains a listing of computer names. The text file, servers.txt, is a plain ASCII text file that has a list of computer names on individual lines, as shown in Figure 14-2.

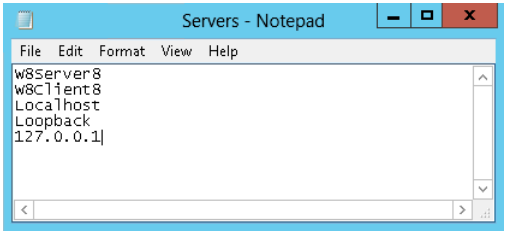


FIGURE 14-2 A text file with computer names and addresses is an easy way to work with allowed computers.

A text file of computer names is easier to maintain than a hard-coded array that is embedded into the script. In addition, the text file can be placed on a central share and can be used by many different scripts. The following example shows the *Get-AllowedComputer* function:

```
Function Get-AllowedComputer([string]$computer)
{
 $servers = Get-Content -path c:\fso\servers.txt
 $servers -contains $computer
} #end Get-AllowedComputer function
```

Because the *Get-AllowedComputer* function returns a Boolean value (*true/false*), it can be used directly in an If statement to determine if the value that is supplied for the *$computer* variable is on the permitted list. If the *Get-AllowedComputer* function returns *true*, the *Get-WmiObject* cmdlet is used to query for BIOS information from the target computer, as show in the following example:

```
if(Get-AllowedComputer -computer $computer)
 {
   Get-WmiObject -class Win32_Bios -Computer $computer
 }
```

On the other hand, if the value of the *$computer* variable is not found in the *$servers* array, a string that states the computer is not an allowed computer is displayed:

```
Else
 {
  "$computer is not an allowed computer"
 }
```

The following example shows the complete Get-AllowedComputer.ps1 script:

```
Get-AllowedComputer.ps1
Param([string]$computer = $env:computername)

Function Get-AllowedComputer([string]$computer)
{
 $servers = Get-Content -path c:\fso\servers.txt
 $servers -contains $computer
} #end Get-AllowedComputer function

# *** Entry point to Script ***

if(Get-AllowedComputer -computer $computer)
 {
   Get-WmiObject -class Win32_Bios -Computer $computer
 }
Else
 {
  "$computer is not an allowed computer"
 }
```

Handling missing rights

Another source of potential errors in a script is one that requires elevated permissions to work correctly. Windows 8 makes it much easier to run and to allow the user to work without requiring constant access to administrative rights. As a result, more and more users and even network administrators are no longer running their computers with a user account that is a member of the local administrators group. The User Account Control (UAC) feature makes it easy to provide elevated rights for interactive programs, but Windows PowerShell 3.0 and other scripting languages are not UAC aware and therefore will not prompt when elevated rights are required to perform a specific activity. It is therefore incumbent upon the script-writer to take rights into account when writing scripts. However, the Get-Bios.ps1 script that we examined earlier in the chapter does not use a WMI class that requires elevated rights. As the script is currently written, anyone who is a member of the local users group, and that includes everyone who is logged on interactively, has permission to run the Get-Bios.ps1 script. So testing for rights and permissions prior to making an attempt to obtain information from the *Win32_Bios* WMI class is not required.

Attempting and failing

One way to handle missing rights is to attempt the action and then fail. This will generate an error. Windows PowerShell has two types of errors: terminating and non-terminating. Terminating errors, as the name implies, will stop a script dead in its tracks. Non-terminating errors will output to the screen and the script will continue. As the names imply, terminating errors are generally more serious than non-terminating errors. Normally, you get a terminating error when you try to use the .NET Framework or the Component Object Model (COM) from within Windows PowerShell and you try to use a command that doesn't exist, or when you do not provide all the required parameters to a command. A good script will handle the errors it expects and will report unexpected errors to the user. Because any good scripting language has to provide decent error handling, Windows PowerShell has a few ways to approach the problem. The old way is to use the *Trap* statement, which can sometimes be problematic. The new way (for Windows PowerShell) is to use the *Try/Catch/Finally* block, which I will cover in the "Using *Try/Catch/Finally*" section later in this chapter.

Checking for rights and exiting gracefully

The best way to handle insufficient rights is to check for the rights and then exit gracefully. What are some of the things that could go wrong with a simple script such as the Get-Bios.ps1 script that we examined earlier in the chapter? Well, the Get-Bios.ps1 script would fail if the Windows PowerShell script execution policy was set to restricted. When the script execution policy is set to restricted, Windows PowerShell scripts will not run. The problem with a restricted execution policy is that because Windows PowerShell scripts do not run, you cannot write code to detect the restricted script execution policy. Because the script execution policy is stored in the registry, you could write a VBScript script that would query and set the policy prior to launching the Windows PowerShell script, but that would not be the best way to manage the problem. The best way to manage the script execution policy is to use Group Policy to set it to the appropriate level for your network. On a stand-alone computer, you can set the execution policy by opening Windows PowerShell as an administrator and using the *Set-ExecutionPolicy* cmdlet. In most cases, the *remotesigned* setting is appropriate. The following example shows the command:

```
PS C:\> Set-ExecutionPolicy remotesigned
PS C:\>
```

The script execution policy is generally dealt with once, and there are no more problems associated with it. In addition, the error message that is associated with the script execution policy is relatively clear in that it will tell you that script execution is disabled on the system. It also refers you to a Help topic that explains the various settings:

```
File C:\Documents and Settings\ed\Local Settings\Temp\tmp2A7.tmp.ps1 cannot be
loaded because the execution of scripts is disabled on this system. Please see
"get-help about_signing" for more details.
At line:1 char:66
+ C:\Documents` and` Settings\ed\Local` Settings\Temp\tmp2A7.tmp.ps1 <<<<
```

Using *Try/Catch/Finally*

When you use a Try/Catch/Finally block, the command you should execute is placed in the Try block. If an error occurs when the command executes, the error will be written to the *$Error* variable and script execution will move to the Catch block. The TestTryCatchFinally.ps1 script uses the *Try* command to attempt to create an object. A string states that the script is attempting to create a new object. The object to create is stored in the *$obj1* variable. The *New-Object* cmdlet creates the object. Once the object has been created and stored in the *$a* variable, the members of the object are displayed by means of the *Get-Member* cmdlet. The following example illustrates the technique:

```
Try
 {
  "Attempting to create new object $obj1"
   $a = new-object $obj1
   "Members of the $obj1"
   "New object $obj1 created"
   $a | Get-Member
 }
```

Use the Catch block to capture errors that occur during the *Try* block. You can specify the type of error to catch as well as the action you wish to perform when the error occurs. The TestTryCatchFinally.ps1 script monitors for *System.Exception* errors. The *System.Exception* .NET Framework class is the base class from which all other exceptions derive. This means a *System.Exception* is as generic as you can get. In essence, it will capture all predefined, common system runtime exceptions. Upon catching the error, you can then specify what code you would like to execute. The following example shows the *Catch* block. A single string states that the script caught a system exception:

```
Catch [system.exception]
 {
  "caught a system exception"
 }
```

The Finally block of a Try/Catch/Finally sequence always runs, regardless if an error is generated or not. This means that any code cleanup you want to do, such as explicitly releasing COM objects, should be placed in a Finally block. In the TestTryCatchFinally.ps1 script, the Finally block displays a string that states the script has ended:

```
Finally
 {
  "end of script"
 }
```

The following example shows the complete TestTryCatchFinally.ps1 script:

```
TestTryCatchFinally.ps1
$obj1 = "Bad.Object"
"Begin test"
Try
 {
  "`tAttempting to create new object $obj1"
   $a = new-object $obj1
   "Members of the $obj1"
   "New object $obj1 created"
   $a | Get-Member
 }
Catch [system.exception]
 {
  "`tcaught a system exception"
 }
Finally
 {
  "end of script"
 }
```

Summary

In this chapter, we focused on handling errors. We began by discussing missing parameters and examining two solutions for detecting and handling parameters. Next, we covered how to limit choices as a means of ensuring quality input to the script. We reviewed how to handle missing rights and concluded by looking at structured error handling.

APPENDIX A

Windows PowerShell FAQ

In this appendix, I cover a range of topics that are organized into a question-and-answer format. I've included questions that frequently arise when I teach a Windows PowerShell class or make Windows PowerShell presentations at various events.

Q. What is Windows PowerShell, in 30 words or less?

A. Windows PowerShell is the next generation command prompt and scripting language from Microsoft. It can be a replacement for VBScript and for the command prompt in most circumstances.

Q. How can you be sure that was 30 words or less?

A. By using the following code:

```
$a = "Windows PowerShell is the next generation command prompt and scripting language
from Microsoft. It can be a replacement for VBScript and for the command prompt in
most circumstances."

Measure-Object -InputObject $a -Word
```

Q. How many cmdlets are available on a default Windows PowerShell installation?

A. 403.

Q. How can I find out how many cmdlets are available on a default Windows PowerShell installation?

```
A. Get-Module -ListAvailable | Import-Module ; gcm -co cmdlet | measure
```

Q. What is the difference between a read-only variable and a constant?

A. A read-only variable is one whose content is read only. It can, however, be modified by using the *Set-Variable* cmdlet with the *-Force* parameter. It can also be deleted by using *Remove-Variable -Force*. However, a constant variable cannot be deleted nor modified, even when using *-Force*.

Q. What are the three MOST important cmdlets?

A. *Get-Command*, *Get-Help*, and *Get-Member*.

Q. Which cmdlet can I use to work with event logs?

A. To work with event logs, use the *Get-Eventlog* cmdlet or the *Get-WinEvent* cmdlet.

Q. How did you find that cmdlet?

A. `Get-Command -Noun *event*`

Q. What .NET Framework class is leveraged by the *Get-Eventlog* cmdlet?

A. *System.Diagnostics.EventLogEntry*.

Q. How do I find the preceding information?

A. `Get-Eventlog application | Get-Member`

Q. What is the most powerful command in Windows PowerShell?

A. *Switch*.

Q. What is *`t* used for?

A. Tab.

Q. How do I use *`t* in a script to produce a tab?

A. *"`thi"*

Q. That syntax above is ugly. What happens if I put a space in it like this?

"`t hi"

A. If you include a space in the line like *"`t hi"*, then you will tab over one tab stop and one additional space.

Q. Is the *`t* command such as *"`thi"* case sensitive?

A. Yes. It is one of the few things that is case sensitive in Windows PowerShell. If you use the *`t* as *"`Thi"*, then you will produce *Thi* on the line.

Q. How do I run a script with a space in the path?

A. There are two options:

```
PS > c:\my`folder\myscript.ps1
```

```
PS> &("c:\my  folder\myscript.ps1")
```

Q. What is the easiest way to create an array?

A. There are two options:

```
$array = "1","2","3","4"
```

```
$array = 1..4
```

Q. How do I display a "calculated value" (in other words, megabytes instead of bytes) from a WMI query when pipelining data into a *Format-Table* cmdlet?

A. Create a hash table in the position where you want to display the data and perform the calculation inside curly brackets. Assign the results to the *Expression* parameter, as shown in the following example:

```
gwmi win32_logicaldisk -Filter "drivetype=3" | ft -Property name, @{ Label="freespace";
expression={$_.freespace/1MB}}
```

Q. Which parameter of the *Get-WMIObject* cmdlet takes the place of a WQL *Where* clause?

A. The filter parameter: `Get-wmiobject win32_logicaldisk -filter "drivetype = 3"`

Q. Which command when typed at the beginning of a script will cause Windows PowerShell to ignore errors and continue executing the code?

A. `$erroractionpreference=SilentlyContinue`

Q. How can I display only the current year?

A. There are three options:

```
get-date -Format yyyy
```

```
get-date -f yyyy
```

```
(Get-Date).year
```

Q. What are three ways of querying Active Directory from within Windows PowerShell?

A. The three ways are the following:

Use ADO and perform an LDAP dialect query.

Use ADO and perform an SQL dialect query.

Use the *Get-ADOObject* cmdlet from the *ActiveDirectory* module.

Q. How can I print the amount of free space on a fixed disk in MB with two decimal places?

A. Use a format specifier as shown in the following example:

```
"{0:n2}"-f ((gwmi win32_logicaldisk -Filter "drivetype='3' ").freespace/1MB)
```

Q. I need to replace the "2" with "12" in the variable *$array*: *$array = "1","2","3","4"*. How can I do this?

A. `$array=[regex]::replace($array, "2","12")`

Q. I have the following *Switch* statement, and I want to prevent the *Write-Host* line "switched" from being executed. How can I do this?

```
$a = 3

switch ($a) {

 1 { "one detected" }

{ "two detected" }

}

Write-Host "switched"
```

A. Add an *Exit* statement to the default switch:

```
$a = 3

switch ($a) {

1 { "one detected" }

2 { "two detected"}

DEFAULT { exit}

}

Write-Host "switched"
```

Q. How can I supply alternative credentials for a remote WMI call when using the *Get-WmiObject* cmdlet?

A. There are several options:

Use the *credential* parameter:

```
Get-WmiObject Win32_BIOS -ComputerName Server01  -Credential (get-credential  `
Domain01@User01)
```

Use the *credential* parameter:

```
$c = Get-Credential

Get-WmiObject Win32_DiskDrive -ComputerName Server01 -Credential $c
```

Create a CIM session to the remote system by using *New-CimSession*.

Create a PS session to the remote system by using *New-PSSession*.

Q. How can I generate a random number?

A. There are two options:

Use the *Get-Random* cmdlet.

Use the *System.Random* .NET Framework class and call the *next()* method:`([random]5).next()`

Q. How can I generate a random number between the values of 1 and 10?

A. There are two options:

Use the *System.Random* .NET Framework class and call the *next()* method:

```
([random]5).next("1","10")
```

Use the *Get-Random* cmdlet: `Get-Random -Maximum 10 -Minimum 1`

Q. Which commands support regular expressions?

A. Two commands support regular expressions:

The *Where-Object* cmdlet using *-match*:

```
get-process | where-object { $_.ProcessName -match "^p.*" }
```

The *Switch* statement using *regex*:

```
switch -regex ("Hi there") { "hi" { "found" } }
```

Q. How can I create an audit file of all commands typed during a Windows PowerShell session?

A. Use the *Start-Transcript* command: `Start-Transcript`

Q. How can I see how many seconds it takes to retrieve objects from the application log?

A. `(Measure-Command { Get-EventLog application }).totalseconds`

Q. I want to get a list of all the modules installed with Windows PowerShell on my machine. How can I do this?

A. Inside a Windows PowerShell console, type the following command:

```
Get-Module -ListAvailable
```

Q. I want to create an ASCII text file to hold the results of the *Get-Process* cmdlet. How can I do this?

A. Pipeline the results to the *Out-File* cmdlet and use the *-Encoding* parameter to specify ASCII. Use redirection like this: `Get-Process >>c:\fso\myprocess.txt`

Q. Someone told me the *Write-Host* cmdlet can use color for output. Can you give me some samples of acceptable syntax?

A. Here are some examples:

```
Write-Host -ForegroundColor 12 "hi"

Write-Host -ForegroundColor 12 "hi" -BackgroundColor white

Write-Host -ForegroundColor blue -BackgroundColor white

Write-Host -ForegroundColor 2 hi

Write-Host -backgroundcolor 2 hi

Write-Host -backgroundcolor ("{0:X}" -f 2) hi

For($i=0 ; $i -le 15 ; $i++) { Write-Host -foregroundcolor $i "hi" }
```

Q. How can I tell if a command completes successfully?

A. There are two options:

Query the *$error* automatic variable. If *$error[0]* reports no information, then no errors have occurred.

Query the *$?* automatic variable. If *$?* is equal to *true,* then the command completed successfully.

Q. How can I split the string in the *$a* variable shown in the following example?

```
$a = "atl-ws-01,atl-ws-02,atl-ws-03,atl-ws-04"
```

A. Use the *split* method: `$b = $a.split(",")`

Q. How do I join an array such as the one in the *$a* variable shown in the following example?

```
$a = "h","e","l","l","o"
```

A. Use the *join* static method from the *string* class: `$b = [string]::join("", $a)`

Q. I need to build a path to the Windows\system32 directory. How can I do this?

A. `Join-Path -path (get-item env:\windir).value -ChildPath system32`

Q. How can I print the value of %systemroot%?

A. There are two options:

```
(get-item Env:\systemroot).value
```

```
$env:systemroot
```

Q. I need to display process output at the Windows PowerShell prompt and write that same output to a text file. How can I do this?

A. `Get-process | Tee-Object -FilePath c:\fso\proc.txt`

Q. I would like to display the *ascii* character associated with the ASCII value *56*. How can I do this?

A. `[char]56`

Q. I want to create a strongly typed array of *system.diagnostics.processes* and store it in a variable called *$a*. How can I do this?

A. `[diagnostics.process[]]$a=get-process`

Q. I want to display the number 1234 in hexadecimal. How can I do this?

A. `"{0:x}" -f 1234`

Q. I want to display the decimal value of the hexadecimal number 0x4d2. How can I do this?

A. `0x4d2`

Q. I want to find out if a string contains the letter *m*. The string is stored in the variable *$a,* as shown in the following example: `$a="northern hairy-nosed wombat"`

A. There are four options:

```
[string]$a.contains("m")

$a.contains("m")

[regex]::match($a,"m")

([regex]::match($a,"m")).success
```

Q. How can I solicit input from the user?

A. Use the *Read-Host* cmdlet: `$in = Read-host "enter the data"`

Q. Can I use a variable named *$input* to hold input from the *Read-Host* cmdlet?

A. The *$input* variable is an automatic variable that is used for script blocks in the middle of a pipeline; as such, it would be a very poor choice. Call the variable *$userInput* or something similar if you want. But do not call it *$input*!

Q. How can I cause the script to generate an error if a variable has not been declared?

A. There are two options:

Place `Set-Psdebug -strict` anywhere in the script. Any non-declared variable will generate an error when accessed.

Use `Set-StrictMode -Version latest.`

Q. How can I increase the size used by the *Get-History* buffer?

A. Assign the desired value to the *$MaximumHistoryCount* automatic variable:

```
$MaximumHistoryCount = 65
```

Q. How can I specify the number 1 as a 16-bit integer array?

A. `$a=[int16[]][int16]1`

Q. I have a string: "*this`"is a string*". I want to replace the " with nothing. No space, just nothing. Effectively, I want to remove the " from the string. The backtick(`) is used here to "escape" the quotation mark. How can I use the *replace* method to replace the " with nothing if the string is held in a variable *$arr*? I want the results to look like this: *thisis a string*

A. There are two options:

Use the *replace* method from the *system.string* .NET Framework class: `$arr.Replace("`"","")`

Use the *ascii* value of the quotation mark, and use the *replace* method from the *system.string* .NET Framework class: `$arr.Replace([char]34,"")`

Q. How can I use *invoke-expression* to run a script inside Windows PowerShell when the path has spaces in it?

A. Escape the spaces with a backtick(`) character and surround it in single quotes:

```
Invoke-Expression ("h:\LABS\extras\Run` With` Spaces.ps1')
```

Q. How can I create an array of byte values that contain hexadecimal values?

A. Use the *[byte]* type constraint, but include the *[]* array character such that the type constraint now looks like *[byte[]]*. To specify a hexadecimal number, use *0x* format. The following example shows the resulting line of code:

```
[byte[]]$mac = 0x00,0x19,0xD2,0x72,0x0E,0x2A
```

Q. How can I count backward?

A. Use a *For* statement. In the second position (the condition), ensure you use greater than or equal for the condition. In the third position (the repeat), use the decrement and assign character, which is a double minus (--). When you put it all together, it will look like the following example:

```
for($i=30;$i -ge 20 ; $i --){$i}
```

APPENDIX B

Windows PowerShell 3.0 coding conventions

This appendix details scripting guidelines. These scripting guidelines have been collected from more than a dozen different scriptwriters from around the world. Most are Microsoft employees actively involved in the world of Windows PowerShell. Some are non-Microsoft employees, such as network administrators and consultants, who use Windows PowerShell on a daily basis to improve their work-life balance. Not every script will adhere to all of these guidelines; however, you will find that the closer you adhere to the guidelines the easier your scripts will be to understand and to maintain. They will not necessarily be easier to write, but they will be easier to manage, and you will find that your total cost of ownership (TCO) on the script should be lowered significantly. In the end, I have only three requirements for a script: that it is easy to read, easy to understand, and easy to maintain.

General script construction

In this section, we cover some general considerations for the overall construction of your scripts. This includes the use of functions and other considerations.

Include functions in the script that uses the functions

While it is possible to use an *include* file or dot-source a function within Windows PowerShell, it can become a support nightmare. If you know which function you want to use, but don't know which script provides it, you have to go looking. If a script provides the function you want but has other elements you don't want, it's hard to pick and choose from the script file. Additionally, you must be very careful when it comes to variable naming conventions as you could end up with conflicting variable names. When you use an *include* file, you no longer have a portable script. It must always travel with the function library.

I use functions in my scripts because it makes the script easier to read and easier to maintain. If I were to store these functions into separate files and then dot-source them, then neither of my two personal objectives of function use is really met.

There is one other consideration: When a script references an external script containing functions, there now exists a relationship that must not be disturbed. For example, if you decide you would like to update the function, you might not remember how many external scripts are calling this function and how it will affect their performance and operation. If

there is only one script calling the function, then the maintenance is easy. However, for only one script, just copy the silly thing into the script file itself and be done with the whole business.

Use full cmdlet names and full parameter names

There are several advantages to spelling out cmdlet names and avoiding the use of aliases in scripts. First of all, it makes your script nearly self-documenting and is therefore much easier to read. Second, it makes the script resilient to alias changes by the user and more compatible with future versions of Windows PowerShell.

Understanding the use of aliases

There are three kinds of aliases in Windows PowerShell: compatibility aliases, canonical aliases, and user-defined aliases.

You can identify the compatibility aliases by using the following command:

```
Get-childitem alias: |

where-object {$_.options -notmatch “Readonly” }
```

The compatibility aliases are present in Windows PowerShell to provide an easier transition from using older command shells. You can remove the compatibility aliases by using the following command:

```
Get-childitem alias: |

where-object {$_.options -notmatch “Readonly” } |

remove-item
```

The canonical aliases were created specifically to make Windows PowerShell cmdlets easier to use from within the Windows PowerShell console. Shortness of length and ease of typing were the primary driving factors in their creation. To find the canonical aliases, use the following command:

```
Get-childitem alias: |

where-object {$_.options -match “Readonly” }
```

If you must use an alias, use only canonical aliases in a script

You are reasonably safe in using canonical aliases in a script. However, they make the script much harder to read, and as there are several aliases for often the same cmdlet, different users of Windows PowerShell might have their own personal favorite alias. Additionally, as the canonical aliases are read-only, even a canonical alias could be removed, or worse yet have the meaning radically altered when the user redefines the alias to have a different meaning.

Always use the *Description* property when creating an alias

When adding aliases to your profile, you might want to specify the read-only or constant option. You should always include the *Description* property for your personal aliases, and make the description something that is relatively constant. Here is an example from my personal Windows PowerShell profile:

```
New-Alias -Name gh -Value Get-Help -Description "mred alias"

New-Alias -Name ga -Value get-alias -Description "mred alias"
```

Use *Get-Item* to convert path strings to rich types

This is actually a pretty cool trick. When working with a listing of files, if you use the *Get-Content* cmdlet, you can only read each line and have it as a path to work with. However, if you use *Get-Item*, you have an object with a corresponding number of both properties and methods to work with. The following example illustrates this:

```
$files = Get-Content "filelist.txt" |

Get-Item $files |

Foreach-object { $_.Fullname }
```

General script readability

One of the most important things you can do when it comes to writing good Windows PowerShell code is to ensure your script is readable. If you can read and understand your Windows PowerShell code, you will avoid 90 percent of all debugging situations and nearly 100 percent of all logic problems. Here are some of my top tips for ensuring your script is readable:

- When creating an alias, include the *-Description* parameter and use it when searching for your personal aliases, as shown in the following example:

```
Get-Alias |

where-object { $_.description -match 'mred' } |

Format-Table -Property "   ",name, definition -autosize `

             -hideTableHeaders
```

- Scripts should accept *-Help* and print a Help text. Use comment-based Help to do this.

- All procedures should begin with a brief comment describing what they do. This description should not describe the implementation details (how it does it) because these often change over time, resulting in unnecessary comment maintenance work or, worse, erroneous comments.
- Arguments passed to a function should be described when their purpose is not obvious and when the function expects the arguments to be in a specific range.
- Return values for variables that are changed by a function should also be described at the beginning of each function.
- Every important variable declaration should include an inline comment describing the use of the variable if the name of the variable is not obvious.
- Variables and functions should be named clearly to ensure that inline comments are needed only for complex functions.
- When creating a complex function with multiple code blocks, place an inline comment for each closing curly bracket (}).
- At the beginning of your script, you should include an overview that describes the script, significant objects and cmdlets, and any unique requirements for the script.
- When naming functions, use the verb and noun construction used by cmdlet names, but avoid the dash in the name. In this way, you can clearly distinguish between the function and the cmdlet. This avoids confusion as to why tab expansion works for one "cmdlet" and not another "cmdlet."
- Scripts should use named parameters if the scripts accept more than one argument. If a script accepts only a single argument, then it is fine to use an unnamed argument.
- Always assume that users will copy your script and modify it to meet their needs. Place comments in the code to facilitate this process.
- Never assume the current path. Always use the full path, either through an environment variable or an explicitly named path.

Formatting your code

Screen space should be conserved as much as possible, while still allowing code formatting to reflect logic structure and nesting. Here are a few suggestions:

- Indent standard nested blocks two spaces.
- Block overview comments for a function.
- Block the highest level statements, with each nested block indented an additional two spaces.
- Line up curly brackets. This will make it easier to follow the code flow.

- Avoid single-line statements. In addition to making it easier to follow the flow of the code, it also makes it easier when you end up searching for a missing curly bracket.
- Break each pipelined object at the pipe. Leave all pipes on the right.
- Avoid line continuation—the back tick (`) character. The exception is when it would cause the user to have to scroll to read the code or the output, which is generally around 90 characters.
- Scripts should follow CamelCase guidelines for long variable names.
- Scripts should use the *Write-Progress* cmdlet if they take more than 1 or 2 seconds to run.
- Consider supporting the *-Whatif* and *-Confirm* parameters in your functions as well as in your scripts, especially if they will change system state. The following example shows use of the *-Whatif* parameter. For a complete script that does this, see the AddNodeEvictNode.ps1 script in Chapter 14.

```
param(
      [switch]$whatif
     )

function funwhatif()
 {
  "what if: Perform operation xxxx"
 }

if($whatif)
  {
   funwhatif #calls the funwhatif() function
  }
```

- If your script does not accept a variable set of arguments, you should check the value of *$args.count* and call the *Help* function if the number is incorrect, as shown in the following example:

```
if($args.count -ge 0)
  {
   "wrong number of arguments"
   Funhelp  #calls the funhelp() function
  }
```

- If your script does not accept any arguments, you should use code such as the following:

```
If($args -ge 0) { funhelp }
```

Working with functions

Functions in Windows PowerShell 3.0 are crucial to both the expandability of Windows PowerShell and the ability to write readable code. As a result, anyone who aspires to become an advanced Windows PowerShell scripter needs to know how to work with functions. Here is a quick list of things to keep in mind:

- Functions should handle mandatory parameter checking. To do this, use parameter property attributes.
- Utility or shared functions can be placed into shared function libraries and then included or dot-sourced into scripts. The file name should be of the form Library-<*noun or featurename*>.ps1, as shown in the following example:

```
. c:\lib\Library-WmiFunctions.ps1
```

- If you are writing a function library script, consider using feature and parameter variable names that incorporate a unique name to minimize the chances of conflict with other variables in the scripts that call them. It is best to store these function libraries in modules to facilitate sharing and use.
- Consider supporting standard parameters when it makes sense for your script. The easiest way to do this is to implement cmdlet binding.

Creating template files

Scripters have used template files to ease their coding needs since the earliest scripting languages. In Windows PowerShell, many people seem to avoid this tried and true technique. However, a few judiciously created template files will yield great dividends in both speed of development and quality of output. Here are some tips to help you to create the best template files:

- Create templates that can be used for different types of scripts. Some examples might be WMI scripts, ADSI scripts, and ADO scripts. When you are creating your templates, consider the following:
  - Add in common functions that you would use on a regular basis.
  - Do not hardcode specific values that the connection strings might require: server names, input file paths, output file paths, and so on. Instead, contain these values in variables.
  - Do not hardcode version information into the template.
  - Make sure you include comments where the template will require modification to be made functional.
  - You might want to turn your templates into code snippets to facilitate their usage.

Writing your own functions

When you write your own functions, there are some practices you might want to consider:

- Create highly specialized functions. Good functions do one thing well.
- Make the function completely self-contained. Good functions should be portable.
- Alphabetize the functions in your script, if possible. This promotes readability and maintainability.
- Give your functions descriptive names, such as *funHelp*, *funLine*, or *funComputePercentage*. I like prefixing my functions with the moniker *fun* to avoid the possibility of running into a keyword and to make them easy to see and read. You can spell out *function*, but I think that is too much typing.
- Every function should have a single output point.
- Every function should have a single entry point.
- Use parameters to avoid problems with local and global variable scopes.
- Implement the common parameters, such as *Verbose*, *Debug*, and *Whatif*. Confirm where appropriate to promote re-usability.

Variables, constants, and naming

Keep the following tips in mind when you create variables, constants, and naming:

- Avoid "magic numbers." When calling methods or functions, avoid hardcoding numeric literals. Instead, create a constant that is descriptive enough that someone reading the code would be able to figure out what the thing is supposed to do. In the ServiceDependencies.ps1 script, we use a number to offset the print out. This number is determined by the position of a certain character in the output. Rather than just saying *+14*, we create a constant with a descriptive name. Refer to Chapter 12 for more information on this script. The following example shows the applicable portion of the code:

```
New-Variable -Name c_padline -value 14 -option constant

Get-WmiObject -Class Win32_DependentService -computername $computer |

Foreach-object `

 {

  "=" * ((([wmi]$_.dependent).pathname).length + $c_padline)
```

- Do not "recycle" variables. These are referred to as un-focused variables. Variables should serve a single purpose. These are called focused variables.
- Give variables descriptive names. Remember you have tab completion, so you should use it to simplify typing.

- Minimize variable scope. If you are going to use a variable only in a function, then declare it in the function.
- When a constant is needed, use a read-only variable instead. Remember that constants cannot be deleted, nor can their value change.
- Avoid hardcoding values into method calls and in the worker section of the script. Instead, place values into variables.
- When possible, group your variables into a single section of each level of the script.
- Avoid using Hungarian notation if it is not needed. Remember everything in Windows PowerShell is basically an object, so there is no value in naming a variable *$objWMI*.
- There are times when it makes sense to use the following: *bln*, *int*, *dbl*, *err*, *dte*, and *str*. This is due to the fact that Windows PowerShell is a strongly typed language. It just acts like it is not.
- Scripts should avoid populating the global variable space. Instead, consider passing values to a function by reference *[ref]*.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Now that you've read the book...

Tell us what you think!

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

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

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