Microsoft

Windows® PowerShell 3.0

Ed Wilson

online book + practice files

Step by Step

Windows PowerShell 3.0 Step by Step

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

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

Discover how to:

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

Your *Step by Step* digital content includes:

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

ISBN: 978-0-7356-6339-8

microsoft.com/mspress

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

Programming/Windows PowerShell

About the Author

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

DEVELOPER ROADMAP

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

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

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

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

Microsoft®

Microsoft

Windows PowerShell™ 3.0 Step by Step

Ed Wilson

ISBN: 978-0-735-66339-8

2 3 4 5 6 7 8 9 10 LSI 8 7 6 5 4 3

Printed and bound in the United States of America.

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

Acquisitions and Developmental Editor: Michael Bolinger

Production Editor: Kristen Borg

Editorial Production: Zyg Group, LLC

Technical Reviewer: Thomas Lee

Copyeditor: Zyg Group, LLC

Indexer: Zyg Group, LLC

Cover Design: Twist Creative • Seattle

Cover Composition: Zyg Group, LLC

Illustrators: Rebecca Demarest and Robert Romano

[2013-04-19]

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

Contents at a Glance

Contents

Foreword

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

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

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

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

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

Introduction

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

Who should read this book

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

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

Assumptions

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

Who should not read this book

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

Organization of this book

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

Finding your best starting point in this book

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

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

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

Conventions and features in this book

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

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

System requirements

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

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

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

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

Code samples

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

http://aka.ms/PowerShellSBS_book

Follow the instructions to download the scripts.zip file.

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

Installing the code samples

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

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

Acknowledgments

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

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

Errata and book support

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

http://www.microsoftpressstore.com/title/ 9780735663398

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

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

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

We want to hear from you

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

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

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

Stay in touch

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

CHAPTER 3

Understanding and Using PowerShell Providers

After completing this chapter, you will be able to:

- Understand the role of providers in Windows PowerShell.
- Use the *Get-PSProvider* cmdlet.
- Use the *Get-PSDrive* cmdlet.
- Use the *New-PSDrive* cmdlet.
- Use the *Get-Item* cmdlet.
- Use the *Set-Location* cmdlet.
- Use the file system model to access data from each of the built-in providers.

Microsoft Windows PowerShell provides a consistent way to access information external to the shell environment. To do this, it uses *providers*. These providers are actually .NET programs that hide all the ugly details to provide an easy way to access information. The beautiful thing about the way the provider model works is that all the different sources of information are accessed in exactly the same manner using a common set of cmdlets—*Get-ChildItem*, for example—to work with different types of data. This chapter demonstrates how to leverage the PowerShell providers.

Note All scripts and files mentioned in this chapter are available via the Microsoft TechNet Script Center (*http://aka.ms/powershellsbs_book*).

Understanding PowerShell providers

By identifying the providers installed with Windows PowerShell, you 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 would connect to the registry "drive" and use the *Get-ChildItem* cmdlet, which is exactly the same method you would use to obtain a listing of files on the hard

drive. The only difference is the specific name associated with each drive. Developers familiar with Windows .NET programming can create new providers, but writing a provider can be complex. (See *http://msdn.microsoft.com/en-us/library/windows/desktop/ee126192(v=vs.85).aspx* for more information.) When a new provider is created, it might ship in a snap-in. A *snap-in* is a *dynamic-link library (DLL)* file that must be installed into Windows PowerShell. After a snap-in has been installed, it cannot be uninstalled unless the developer provides removal logic—however, the snap-in can be removed from the current Windows PowerShell console. The preferred way to ship a provider is via a Windows PowerShell module. Modules are installable via an Xcopy deployment, and therefore do not necessarily require admin rights.

To obtain a listing of all the providers, use the *Get-PSProvider* cmdlet. This command produces the following list on a default installation of Windows PowerShell (Windows 8 does not include the *WSMan* provider):

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

Understanding the alias provider

In Chapter 1, "Overview of Windows PowerShell 3.0," I presented 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. 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.

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

Working with the alias provider

1. Open the Windows PowerShell console.
2. Obtain a listing of all the providers by using the *Get-PSProvider* cmdlet.

3. The PowerShell drive (PS drive) associated with the alias provider is called Alias. This is shown in the listing produced by the *Get-PSProvider* cmdlet. Use the *Set-Location* cmdlet to change to the Alias drive. Use the *sl* alias to reduce typing. This command is shown here:

```
sl alias:\
```

4. Use the *Get-ChildItem* cmdlet to produce a listing of all the aliases that are defined on the system. To reduce typing, use the alias *gci* in place of *Get-ChildItem*. This is shown here:

```
gci
```

5. Use a *Where-Object* cmdlet filter to reduce the amount of information that is returned by using the *Get-ChildItem* cmdlet. Produce a listing of all the aliases that begin with the letter s. This is shown here:

```
gci | Where name -like "s*"
```

6. To identify other properties that could be used in the filter, pipeline the results of the *Get-ChildItem* cmdlet into the *Get-Member* cmdlet. This is shown here (keep in mind that different providers expose different objects that will have different properties):

```
Get-ChildItem | Get-Member
```

7. Press the up arrow key twice, and edit the previous filter to include only definitions that contain the word *set*. The modified filter is shown here:

```
gci | Where definition -like "set*"
```

8. The results of this command are shown here:

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           cd -> Set-Location
Alias           chdir -> Set-Location
Alias           sal -> Set-Alias
Alias           sbp -> Set-PSBreakpoint
Alias           sc -> Set-Content
Alias           set -> Set-Variable
Alias           si -> Set-Item
Alias           sl -> Set-Location
Alias           sp -> Set-ItemProperty
Alias           sv -> Set-Variable
Alias           swmi -> Set-WmiInstance
```

9. Press the up arrow key three times, and edit the previous filter to include only names of aliases that are like the letter *w*. This revised command is shown here:

```
gci | Where name -like "*w*"
```

The results from this command will be similar to those shown here:

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           fw -> Format-Wide
Alias           gwmi -> Get-WmiObject
Alias           iwmi -> Invoke-WmiMethod
Alias           iwr -> Invoke-WebRequest
Alias           pwd -> Get-Location
Alias           rwmi -> Remove-WmiObject
Alias           swmi -> Set-WmiInstance
Alias           where -> Where-Object
Alias           wjb -> Wait-Job
Alias           write -> Write-Output
```

10. In the preceding list, note that *where* is an alias for the *Where-Object* cmdlet. Press the up arrow key one time to retrieve the previous command. Edit it to use the *where* alias instead of spelling out the entire *Where-Object* cmdlet name. This revised command is shown here:

```
gci | where name -like "*w*"
```

Caution When using the *Set-Location* cmdlet to switch to a different PS drive, you must follow the name of the PS drive with a colon. A trailing forward slash or backward slash is optional. An error will be generated if the colon is left out, as shown in Figure 3-1. I prefer to use the backward slash (\) because it is consistent with normal Windows file system operations.

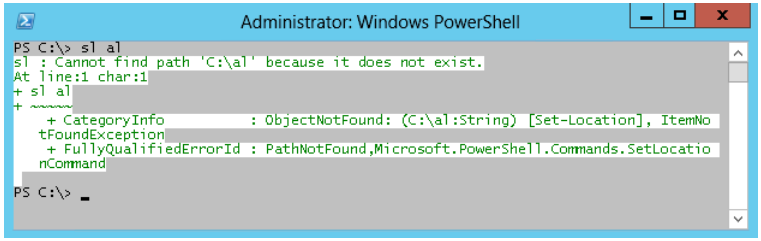

FIGURE 3-1 Using *Set-Location* without a colon results in an error.

Understanding the certificate provider

The preceding section explored working with the alias provider. Because the file system model applies to the certificate provider in much the same way as it does the alias provider, many of the same cmdlets can be used. To find information about the certificate provider, use the *Get-Help* cmdlet and search for *about_Providers*. If you are unsure what articles in help may be related to certificates, you can use the wildcard asterisk (*) parameter. This command is shown here:

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

In addition to allowing you to use the certificate provider, Windows PowerShell gives you the ability to sign scripts; Windows PowerShell can work with signed and unsigned scripts as well. The certificate provider gives you the ability search for, copy, move, and delete certificates. Using the certificate provider, you can open the Certificates Microsoft Management Console (MMC). The commands used in the following procedure use the certificate provider to obtain a listing of the certificates installed on the local computer.

Obtaining a listing of certificates

1. Open the Windows PowerShell console.
2. Set your location to the cert PS drive. To do this, use the *Set-Location* cmdlet, as shown here:

```
Set-Location cert:\
```

3. Use the *Get-ChildItem* cmdlet to produce a list of the certificates, as shown here:

```
Get-ChildItem
```

   The list produced is shown here:

```
Location   : CurrentUser
StoreNames : {?, UserDS, AuthRoot, CA...}

Location   : LocalMachine
StoreNames : {?, AuthRoot, CA, AddressBook...}
```

4. Use the *-recurse* argument to cause the *Get-ChildItem* cmdlet to produce a list of all the certificate stores and the certificates in those stores. To do this, press the up arrow key one time and add the *-recurse* argument to the previous command. This is shown here:

```
Get-ChildItem -recurse
```

5. Use the *-path* argument for *Get-ChildItem* to produce a listing of certificates in another store, without using the *Set-Location* cmdlet to change your current location. Use the *gci* alias, as shown here:

```
GCI -path currentUser
```

   Your listing of certificate stores will look similar to the one shown here:

```
Name : ?

Name : UserDS

Name : AuthRoot

Name : CA

Name : AddressBook
```

```
Name : ?

Name : Trust

Name : Disallowed

Name : _NMSTR

Name : ?????k

Name : My

Name : Root

Name : TrustedPeople

Name : ACRS

Name : TrustedPublisher

Name : REQUEST
```

6. Change your working location to the currentuser\authroot certificate store. To do this, use the *sl* alias followed by the path to the certificate store (*sl* is an alias for the *Set-Location* cmdlet). This command is shown here:

```
sl currentuser\authroot
```

7. Use the *Get-ChildItem* cmdlet to produce a listing of certificates in the currentuser\authroot certificate store that contain the name *C&W* in the subject field. Use the *gci* alias to reduce the amount of typing. Pipeline the resulting object to a *Where-Object* cmdlet, but use the *where* alias instead of typing *Where-Object*. The code to do this is shown here:

```
gci | where subject -like "*c&w*"
```

On my machine, there are four certificates listed. These are shown here:

```
Thumbprint                                Subject
----------                                -------
F88015D3F98479E1DA553D24FD42BA3F43886AEF  O=C&W HKT SecureNet CA SGC Root, C=hk
9BACF3B664EAC5A17BED08437C72E4ACDA12F7E7  O=C&W HKT SecureNet CA Class A, C=hk
4BA7B9DDD68788E12FF852E1A024204BF286A8F6  O=C&W HKT SecureNet CA Root, C=hk
47AFB915CDA26D82467B97FA42914468726138DD  O=C&W HKT SecureNet CA Class B, C=hk
```

8. Use the up arrow key, and edit the previous command so that it will return only certificates that contain the phrase *SGC Root* in the subject property. The revised command is shown here:

```
gci | where subject -like "*SGC Root*"
```

9. The resulting output on my machine contains an additional certificate. This is shown here:

```
Thumbprint                                Subject
----------                                -------
F88015D3F98479E1DA553D24FD42BA3F43886AEF  O=C&W HKT SecureNet CA SGC Root, C=hk
687EC17E0602E3CD3F7DFBD7E28D57A0199A3F44  O=SecureNet CA SGC Root, C=au
```

10. Use the up arrow key and edit the previous command. This time, change the *Where-Object* cmdlet so that it filters on the thumbprint attribute that is equal to *F88015D3F98479E1DA553D24FD42BA3F43886AEF*. You do not have to type that, however; to copy the thumbprint, you can highlight it and press Enter in Windows PowerShell, as shown in Figure 3-2. The revised command is shown here:

```
gci | where thumbprint -eq "F88015D3F98479E1DA553D24FD42BA3F43886AEF"
```

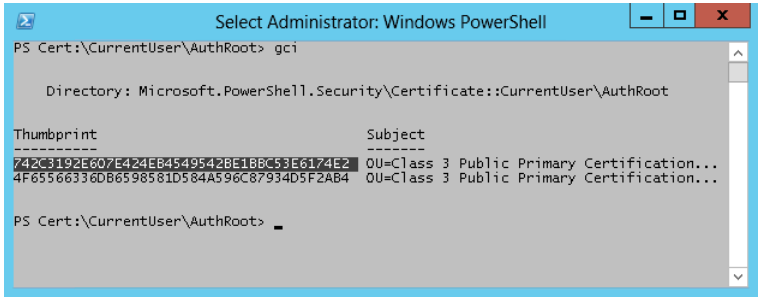

FIGURE 3-2 Highlight items to copy using the mouse.

Troubleshooting If copying from inside the Windows PowerShell console window does not work, then you may need to enable QuickEdit mode. To do this, right-click the PowerShell icon in the upper-left corner of the Windows PowerShell window. Choose Properties, click the Options tab, and then select QuickEdit Mode. This is shown in Figure 3-3.

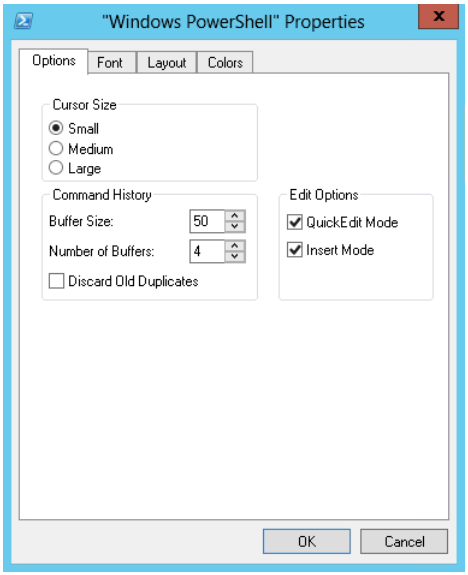

FIGURE 3-3 Enable QuickEdit mode to enable clipboard support.

11. To see all the properties of the certificate, pipeline the certificate object to a *Format-List* cmdlet and choose all the properties. The revised command is shown here:

```
gci | where thumbprint -eq " E0AB05942072549305606202367OF7CD2EFC6666" |
Format-List *
```

The output contains all the properties of the certificate object and is shown here:

```
PSPath                  : Microsoft.PowerShell.Security\Certificate::currentuser\
                          authroot\E0AB05942072549305606202367OF7CD2EFC6666
PSParentPath            : Microsoft.PowerShell.Security\Certificate::currentuser\
                          authroot
PSChildName             : E0AB05942072549305606202367OF7CD2EFC6666
PSDrive                 : Cert
PSProvider              : Microsoft.PowerShell.Security\Certificate
PSIsContainer           : False
EnhancedKeyUsageList    : {Server Authentication (1.3.6.1.5.5.7.3.1), Code Signing
                          (1.3.6.1.5.5.7.3.3), Time Stamping (1.3.6.1.5.5.7.3.8)}
DnsNameList             : {Thawte Premium Server CA}
```

```
SendAsTrustedIssuer      : False
EnrollmentPolicyEndPoint : Microsoft.CertificateServices.Commands.EnrollmentEndPoint
                           Property
EnrollmentServerEndPoint : Microsoft.CertificateServices.Commands.EnrollmentEndPoint
                           Property
PolicyId                 :
Archived                 : False
Extensions               : {System.Security.Cryptography.Oid}
FriendlyName             : Thawte Premium Server CA (SHA1)
IssuerName               : System.Security.Cryptography.X509Certificates.X500
                           DistinguishedName
NotAfter                 : 1/1/2021 6:59:59 PM
NotBefore                : 7/31/1996 8:00:00 PM
HasPrivateKey            : False
PrivateKey               :
PublicKey                : System.Security.Cryptography.X509Certificates.PublicKey
RawData                  : {48, 130, 3, 54...}
SerialNumber             : 36122296C5E338A520A1D25F4CD70954
SubjectName              : System.Security.Cryptography.X509Certificates.X500
                           DistinguishedName
SignatureAlgorithm       : System.Security.Cryptography.Oid
Thumbprint               : E0AB05942072549305606202367OF7CD2EFC6666
Version                  : 3
Handle                   : 647835770000
Issuer                   : E=premium-server@thawte.com, CN=Thawte Premium Server
                           CA, OU=Certification Services Division, O=Thawte
                           Consulting cc, L=Cape Town, S=Western Cape, C=ZA
Subject                  : E=premium-server@thawte.com, CN=Thawte Premium Server
                           CA, OU=Certification Services Division, O=Thawte
                           Consulting cc, L=Cape Town, S=Western Cape, C=ZA
```

12. Open the Certificates MMC file. This MMC file is called Certmgr.msc; you can launch it by simply typing the name inside Windows PowerShell, as shown here:

```
Certmgr.msc
```

13. But it is more fun to use the *Invoke-Item* cmdlet to launch the Certificates MMC. To do this, supply the PS drive name of cert:\ to the *Invoke-Item* cmdlet. This is shown here:

```
Invoke-Item cert:\
```

14. Compare the information obtained from Windows PowerShell with the information displayed in the Certificates MMC. It should be the same. The certificate is shown in Figure 3-4.

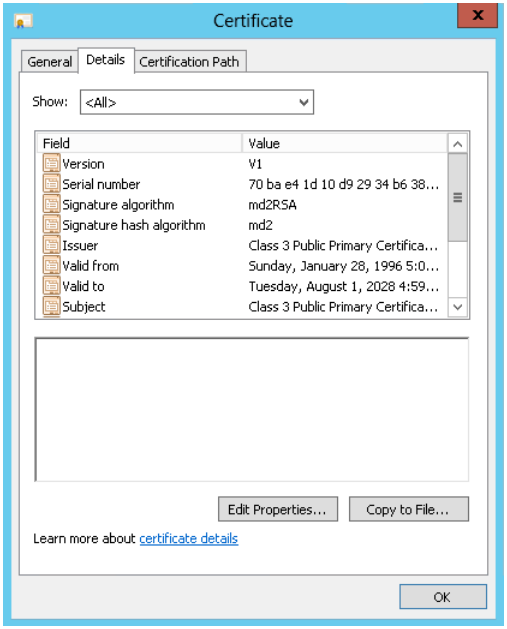

FIGURE 3-4 Certmgr.msc can be used to examine certificate properties.

This concludes this procedure.

Searching for specific certificates

To search for specific certificates, you may 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. This command and associated output appear here:

```
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 simply requires pipelining the results of the previous command to the *Remove-Item* cmdlet.

Note When performing any operation that may 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 of the certificates from the currentuser store that contain the word *test* in the *subject* property. Once completed, retrieve the command via the up arrow key and remove the *-whatif* switched parameter from the command prior to actual execution. This technique appears here:

```
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 either have expired or are about to expire. Using the certificate provider, it is simple to identify 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. This technique appears here:

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

A more flexible approach is to use the current date—therefore, each time the command runs, it retrieves expired certificates. This technique appears here:

```
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 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 upon date, you can no longer use the simple *Where-Object* syntax. The *$_* character represents the current certificate as it comes across the pipeline. Because you're comparing two properties, you must repeat the *$_* character for each property. The following command retrieves the expiration dates, thumbprints, and 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 pipe 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
```

Caution 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—if you do, you could cause serious damage to your system.

If you want to identify certificates that will expire in the next 30 days, you use the same technique involving a compound *Where-Object* command. The command appearing here identifies certificates expiring in the next 30 days:

```
PS Cert:\> dir .\\CurrentUser -Recurse |
where { $_.NotAfter -gt (Get-Date) -AND $_.NotAfter -le (Get-Date).Add(30) }
```

Understanding the environment provider

The environment provider in Windows PowerShell is used to provide access to the system environment variables. If you open a CMD (command) shell and type **set**, you will obtain a listing of all the 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 CMD prompt when you are inside of the Windows PowerShell console. Typing **exit** returns you to Windows PowerShell. The best way to determine whether you are running the command shell or Windows PowerShell is to examine the prompt. The default Windows PowerShell prompt is PS C:\>, assuming that you are working on drive C.

If you use the *echo* command in the CMD interpreter to print out the value of *%windir%*, you will obtain the results shown in Figure 3-5.

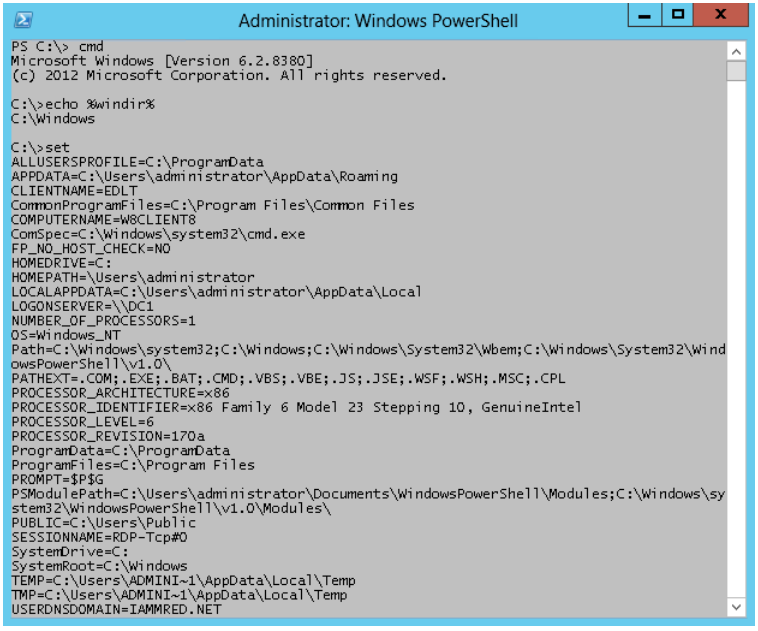

FIGURE 3-5 Use *set* at a CMD prompt to see environment variables.

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.

Obtaining a listing of environment variables

1. Open the Windows PowerShell console.
2. Obtain a listing of the PS drives by using the *Get-PSDrive* cmdlet. This is shown here:

```
Get-PSDrive
```

3. Note that the Environment PS drive is called *Env*. Use the *Env* name with the *Set-Location* cmdlet and change to the Environment PS drive. This is shown here:

```
Set-Location Env:\
```

4. Use the *Get-Item* cmdlet to obtain a listing of all the environment variables on the system. This is shown here:

```
Get-Item *
```

5. Use the *Sort-Object* cmdlet to produce an alphabetical listing of all the environment variables by name. Use the up arrow key to retrieve the previous command, and pipeline the returned object into the *Sort-Object* cmdlet. Use the *-property* argument, and supply *name* as the value. This command is shown here:

```
Get-Item * | Sort-Object  -property value
```

6. Use the *Get-Item* cmdlet to retrieve the value associated with the environment variable *windir*. This is shown here:

```
Get-Item windir
```

7. Use the up arrow key and retrieve the previous command. Pipeline the object returned to the *Format-List* cmdlet and use the wildcard character to print out all the properties of the object. The modified command is shown here:

```
Get-Item windir | Format-List *
```

8. The properties and their associated values are shown here:

```
PSPath        : Microsoft.PowerShell.Core\Environment::windir
PSDrive       : Env
PSProvider    : Microsoft.PowerShell.Core\Environment
PSIsContainer : False
Name          : windir
Key           : windir
Value         : C:\WINDOWS
```

This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Creating a temporary new environment variable

1. You should still be in the Environment PS drive from the previous procedure. If not, use the *Set-Location env:*\ command).

2. Use the *Get-Item* cmdlet to produce a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet using the property *name*. To reduce typing, use the *gi* alias and the *sort* alias. This is shown here:

```
gi * | sort -property name
```

3. Use the *New-Item* cmdlet to create a new environment variable. The *-path* argument will be dot (.) because you are already on the env:\ PS drive. The *-Name* argument will be *admin*, and the *-value* argument will be your given name. The completed command is shown here:

```
New-Item -Path . -Name admin -Value mred
```

4. Use the *Get-Item* cmdlet to ensure the *admin* environment variable was properly created. This command is shown here:

```
Get-Item admin
```

The results of the previous command are shown here:

```
Name                           Value
----                           -----
admin                          mred
```

5. Use the up arrow key to retrieve the previous command. Pipeline the results to the *Format-List* cmdlet and choose All Properties. This command is shown here:

```
Get-Item admin | Format-List *
```

The results of the previous command include the PS path, PS drive, and additional information about the newly created environment variable. These results are shown here:

```
PSPath        : Microsoft.PowerShell.Core\Environment::admin
PSDrive       : Env
PSProvider    : Microsoft.PowerShell.Core\Environment
PSIsContainer : False
Name          : admin
Key           : admin
Value         : mred
```

The new environment variable exists until you close the Windows PowerShell console.

This concludes this procedure. Leave PowerShell open for the next procedure.

Renaming an environment variable

1. Use the *Get-ChildItem* cmdlet to obtain a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet and sort the list on the *name* property. Use the *gci* and *sort* aliases to reduce typing. The code to do this is shown here:

   ```
   gci | sort -property name
   ```

2. The *admin* environment variable should be near the top of the list of system variables. If it is not, then create it by using the *New-Item* cmdlet. The *-path* argument has a value of dot (.); the *-name* argument has the value of *admin*, and the *-value* argument should be the user's given name. If this environment variable was created in the previous exercise, then PowerShell will report that it already exists. The command appearing here allows you to re-create the *admin* environment variable:

   ```
   New-Item -Path . -Name admin -Value mred
   ```

3. Use the *Rename-Item* cmdlet to rename the *admin* environment variable to *super*. The *-path* argument combines the PS drive name with the environment variable name. The *-NewName* argument is the desired new name without the PS drive specification. This command is shown here:

   ```
   Rename-Item -Path env:admin -NewName super
   ```

4. To verify that the old environment variable *admin* has been renamed *super*, press the up arrow key two or three times to retrieve the *gci* | *sort -property name* command. This command is shown here:

   ```
   gci | sort -property name
   ```

   This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Removing an environment variable

1. Use the *Get-ChildItem* cmdlet to obtain a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet and sort the list on the *name* property. Use the *gci* and *sort* aliases to reduce typing. The code to do this is shown here:

   ```
   gci | sort -property name
   ```

2. The *super* environment variable should be in the list of system variables. If it is not, then create it by using the *New-Item* cmdlet. The *-path* argument has a value of dot (.), the *-name* argument has a value of *super*, and the *-value* argument should be the user's given name. If this environment variable was created in the previous exercise, then PowerShell will report that it already exists. If you have deleted the *admin* environment variable, the command appearing here creates it:

   ```
   New-Item -Path . -Name super -Value mred
   ```

3. Use the *Remove-Item* cmdlet to remove the *super* environment variable. The name of the item to be removed is typed following the name of the cmdlet. If you are still in the env:\ PS drive, you will not need to supply a *-path* argument. The command is shown here:

   ```
   Remove-Item env:super
   ```

4. Use the *Get-ChildItem* cmdlet to verify that the environment variable *super* has been removed. To do this, press the up arrow key two or three times to retrieve the *gci | sort -property name* command. This command is shown here:

   ```
   gci | sort -property name
   ```

   This concludes this procedure.

Understanding the filesystem provider

The filesystem provider is the easiest Windows PowerShell provider to understand—it provides access to the file system. When Windows PowerShell is launched, it automatically opens on the user documents folder. Using the Windows PowerShell filesystem 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. The commands used in the procedure are in the IdentifyingPropertiesOfDirectories.txt, CreatingFoldersAndFiles.txt, and ReadingAndWritingForFiles.txt files and are available from the Technet Script Repository, at *http://aka.ms/powershellSBS_book*.

Working with directory listings

1. Open the Windows PowerShell console.
2. Use the *Get-ChildItem* cmdlet to obtain a directory listing of drive C. Use the *gci* alias to reduce typing. This is shown here:

```
GCI C:\
```

3. Use the up arrow key to retrieve the *gci C:* command. Pipeline the object created into a *Where-Object* cmdlet and look for containers. This will reduce the output to only directories. The modified command is shown here:

```
GCI C:\ | where psiscontainer
```

4. Use the up arrow key to retrieve the *gci C:\ | where psiscontainer* command, and use the exclamation point (!) (meaning *not*) to retrieve only items in the PS drive that are not directories. The modified command is shown here. (The simplified *Where-Object* syntax does not support using the *not* operator directly on the input property.)

```
gci | ? {!($psitem.psiscontainer)}
```

This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Identifying properties of directories

1. Use the *Get-ChildItem* cmdlet and supply a value of *C:* for the *-path* argument. Pipeline the resulting object into the *Get-Member* cmdlet. Use the *gci* and *gm* aliases to reduce typing. This command is shown here:

```
gci  -path C:\ | gm
```

2. The resulting output contains methods, properties, and more. Filter the output by pipelining it into a *Where-Object* cmdlet and specifying the *membertype* attribute as equal to *property*. To do this, use the up arrow key to retrieve the previous *gci -path C:\ | gm* command. Pipeline the resulting object into the *Where-Object* cmdlet and filter on the *membertype* attribute. The resulting command is shown here:

```
gci  -path C:\ | gm | Where {$_.membertype -eq "property"}
```

3. On Windows 8, you need to use the *-force* parameter to see hidden files. Here is the command:

```
gci  -path C:\ -force | gm | Where {$_.membertype -eq "property"}
```

4. The preceding *gci -path C:\ | gm | where {$_.membertype -eq "property"}* command returns information on both the *System.IO.DirectoryInfo* and *System.IO.FileInfo* objects (on Windows 8, you need to use the *-force* switch to see hidden files). To reduce the output to only the properties associated with the *System.IO.FileInfo* object, you need to use a compound *Where-Object* cmdlet. Use the up arrow key to retrieve the *gci -path C:\ | gm | where {$_.membertype -eq "property"}* command. Add the *And* conjunction and retrieve objects that have a type name that is like **file**. The modified command is shown here:

```
gci  -path C:\ | gm |
where {$_.membertype -eq "property" -AND $_.typename -like  "*file*"}
```

5. On Windows 8, you need to use the *-force* parameter. Here is the command to do that:

```
gci  -path C:\ -force | gm |
where {$_.membertype -eq "property" -AND $_.typename -like  "*file*"}
```

6. The resulting output contains only the properties for a *System.IO.FileInfo* object. These properties are shown here:

```
TypeName: System.IO.FileInfo

Name              MemberType Definition
----              ---------- ----------
Attributes        Property   System.IO.FileAttributes Attributes {get;set;}
CreationTime      Property   System.DateTime CreationTime {get;set;}
CreationTimeUtc   Property   System.DateTime CreationTimeUtc {get;set;}
Directory         Property   System.IO.DirectoryInfo Directory {get;}
DirectoryName     Property   System.String DirectoryName {get;}
Exists            Property   System.Boolean Exists {get;}
Extension         Property   System.String Extension {get;}
FullName          Property   System.String FullName {get;}
IsReadOnly        Property   System.Boolean IsReadOnly {get;set;}
LastAccessTime    Property   System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc Property   System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime     Property   System.DateTime LastWriteTime {get;set;}
LastWriteTimeUtc  Property   System.DateTime LastWriteTimeUtc {get;set;}
Length            Property   System.Int64 Length {get;}
Name              Property   System.String Name {get;}
```

This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Creating folders and files

1. Use the *Get-Item* cmdlet to obtain a listing of files and folders. Pipeline the resulting object into the *Where-Object* cmdlet and use the *PsisContainer* property to look for folders. Use the name property to find names that contain the word *my* in them. Use the *gi* alias and the *where* alias to reduce typing. The command is shown here:

```
Set-Location c:\Mytest
GI * | Where {$_.PsisContainer -AND $_.name -Like "*my*"}
```

2. If you were following along in the previous chapters, you will have a folder called Mytest off the root of drive C. Use the *Remove-Item* cmdlet to remove the Mytest folder. Specify the *-recurse* argument to also delete files contained in the C:\Mytest folder. If your location is still set to Env, then change it to C or search for C:\Mytest. The command is shown here:

```
RI mytest -recurse
```

3. Press the up arrow key twice and retrieve the *gi * | where {$_.PsisContainer -AND $_.name -Like "*my*"}* command to confirm the folder was actually deleted. This command is shown here:

```
gi * | where {$_.PsisContainer -AND $_.name -Like "*my*"}
```

4. Use the *New-Item* cmdlet to create a folder named Mytest. Use the *-path* argument to specify the path of C:\. Use the *-name* argument to specify the name of Mytest, and use the *-type argument* to tell Windows PowerShell the new item will be a directory. This command is shown here:

```
New-Item -Path C:\ -name mytest -type directory
```

The resulting output, shown here, confirms the operation:

```
 Directory: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----          5/4/2012   2:43 AM            mytest
```

5. Use the *New-Item* cmdlet to create an empty text file. To do this, use the up arrow key and retrieve the previous *New-Item -path C:\ -name Mytest -type directory* command. Edit the *-path* argument so that it is pointing to the C:\Mytest directory. Edit the *-name* argument to specify a text file named Myfile, and specify the *-type* argument as *file*. The resulting command is shown here:

```
New-Item -path C:\mytest -name myfile.txt -type file
```

The resulting message, shown here, confirms the creation of the file:

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\mytest

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          5/4/2012   3:12 AM          0 myfile.txt
```

This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Reading and writing for files

1. Delete Myfile.txt (created in the previous procedure). To do this, use the *Remove-Item* cmdlet and specify the *-path* argument as C:\Mytest\Myfile.txt. This command is shown here:

```
RI -Path C:\mytest\myfile.txt
```

2. Use the up arrow key twice to retrieve the *New-Item -path C:\Mytest -name Myfile.txt -type* command. Add the *-value* argument to the end of the command line and supply a value of *My file*. This command is shown here:

```
New-Item -Path C:\mytest -Name myfile.txt -Type file -Value "My file"
```

3. Use the *Get-Content* cmdlet to read the contents of myfile.txt. This command is shown here:

```
Get-Content C:\mytest\myfile.txt
```

4. Use the *Add-Content* cmdlet to add additional information to the myfile.txt file. This command is shown here:

```
Add-Content C:\mytest\myfile.txt -Value "ADDITIONAL INFORMATION"
```

5. Press the up arrow key twice and retrieve the *Get-Content C:\mytest\myfile.txt* command, which is shown here:

```
Get-Content C:\mytest\myfile.txt
```

6. The output from the *Get-Content C:\mytest\myfile.txt* command is shown here:

```
My fileADDITIONAL INFORMATION
```

7. Press the up arrow key twice, and retrieve the *Add-Content C:\mytest\myfile.txt -value "ADDITIONAL INFORMATION"* command to add additional information to the file. This command is shown here:

```
Add-Content C:\mytest\myfile.txt -Value "ADDITIONAL INFORMATION"
```

8. Use the up arrow key to retrieve the *Get-Content C:\mytest\myfile.txt* command, which is shown here:

```
Get-Content C:\mytest\myfile.txt
```

9. The output produced is shown here. Notice that the second time the command runs, the *"ADDITIONAL INFORMATION"* string is added to a new line in the original file.

```
My fileADDITIONAL INFORMATION
ADDITIONAL INFORMATION
```

10. Use the *Set-Content* cmdlet to overwrite the contents of the Myfile.txt file. Specify the *-value* argument as *Setting information*. This command is shown here:

```
Set-Content C:\mytest\myfile.txt -value "Setting information"
```

11. Use the up arrow key to retrieve the *Get-Content C:\Mytest\Myfile.txt* command, which is shown here:

```
Get-Content C:\mytest\myfile.txt
```

The output from the *Get-Content* command is shown here:

```
Setting information
```

This concludes this procedure.

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 the functions on your system. You can also add, modify, and delete functions. The function provider uses a file system–based model, and the cmdlets described earlier apply to working with functions. The commands used in the following procedure are in the ListingAllFunctionsOnTheSystem.txt file.

Listing all functions on the system

1. Open the Windows PowerShell console.
2. Use the *Set-Location* cmdlet to change the working location to the Function PS drive. This command is shown here:

```
Set-Location function:\
```

3. Use the *Get-ChildItem* cmdlet to enumerate all the functions. Do this by using the *gci* alias, as shown here:

```
gci
```

4. The resulting list contains many functions that use *Set-Location* to change the current location to different drive letters. A partial view of this output is shown here:

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        A:
Function        B:
Function        C:
Function        cd..
Function        cd\
Function        Clear-Host
<truncated...>
```

```
Function        Get-Verb
Function        H:
Function        help
Function        I:
Function        ImportSystemModules
<truncated...>
Function        mkdir
Function        more
Function        N:
Function        O:
Function        oss
Function        P:
Function        Pause
Function        prompt
<truncated ...>
Function        TabExpansion2
<truncated ...>
```

5. To return only the functions that are used for drives, use the *Get-ChildItem* cmdlet and pipe the object returned into a *Where-Object* cmdlet. Use the default *$_* variable to filter on the *definition* attribute. Use the *-like* argument to search for definitions that contain the word *set*. The resulting command is shown here:

```
gci | Where definition -like "set*"
```

6. If you are more interested in functions that are not related to drive mappings, then you can use the *-notlike* argument instead of *-like*. The easiest way to make this change is to use the up arrow key and retrieve the *gci | where {$_.definition -like "set*"}* command, and then change the filter from *-like* to *-notlike*. The resulting command is shown here:

```
gci | Where definition -notlike "set*"
```

The resulting listing of functions is shown here:

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Clear-Host
Function        Get-Verb
Function        help
Function        ImportSystemModules
Function        mkdir
Function        more
Function        oss
Function        Pause
Function        prompt
Function        TabExpansion2
```

7. Use the *Get-Content* cmdlet to retrieve the text of the *pause* function. This is shown here (*gc* is an alias for the *Get-Content* cmdlet):

```
gc pause
```

The content of the *pause* function is shown here:

```
Read-Host 'Press Enter to continue...' | Out-Null
```

This concludes this procedure.

Using the registry provider to manage the Windows registry

In Windows PowerShell 1.0, the registry provider made it easy to work with the registry on the local system. Unfortunately, without remoting, you were limited to working with the local computer or using some other remoting mechanism (perhaps a log-on 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.

The registry provider permits access to the registry in the same manner that the filesystem provider permits access to a local disk drive. The same cmdlets used to access the file system—*New-Item*, *Get-ChildItem*, *Set-Item*, *Remove-Item*, and so on—also work with the registry.

The two registry drives

By default, the registry provider creates two registry drives. To find all of the drives exposed by the registry provider, use the *Get-PSDrive* cmdlet. These drives appear here:

```
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. The code to do this appears here:

```
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*). This technique appears here:

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

To determine the current location, use the *Get-Location* cmdlet. This technique appears here:

```
PS HKCR:\> Get-Location

Path
----
HKCR:\
```

Once you've set the new working location, explore it by using the *Get-ChildItem* cmdlet (or one of the aliases for that cmdlet, such as *dir*). This technique appears in Figure 3-6.

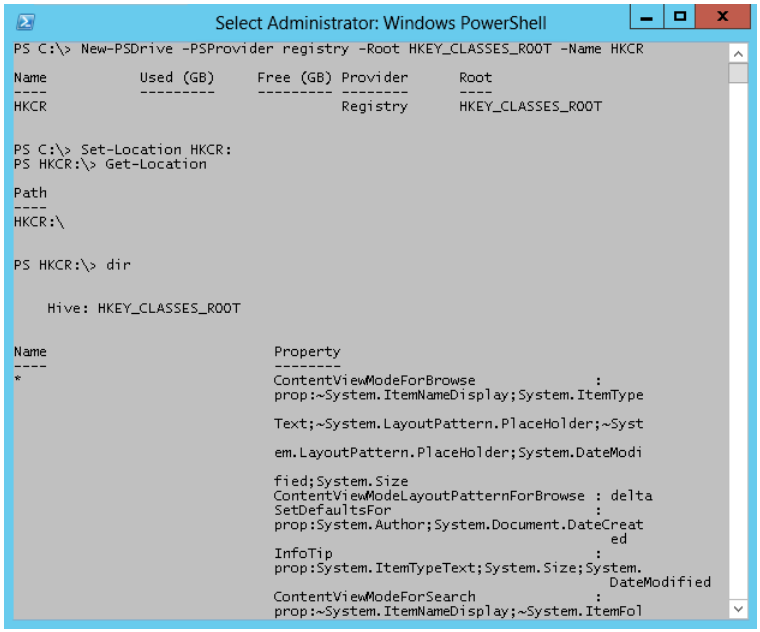

FIGURE 3-6 Creating a new registry drive for the HKEY_CLASSES_ROOT registry hive enables easy access to class registration information.

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*). This appears here:

```
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, as shown here:

```
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
```

The technique for accessing registry keys and the values associated with them appears in Figure 3-7.

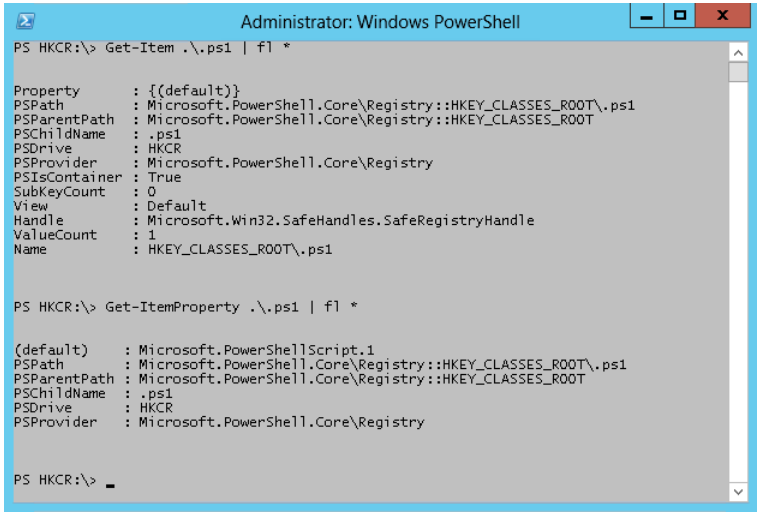

FIGURE 3-7 Use the *Get-ItemProperty* cmdlet to access registry property values.

Returning 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. This appears here:

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

The registry provider provides a consistent and easy way to work with the registry from within Windows PowerShell. Using the registry provider, you can search the registry, create new registry keys, delete existing registry keys, and modify values and access control lists (ACLs) from within Windows PowerShell.

The commands used in the following procedure are in the UnderstandingTheRegistryProvider.txt file. Two PS drives are created by default. To identify the PS drives 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*. This command is shown here:

```
PS C:\> Get-PSDrive | ? provider -match registry

Name           Used (GB)     Free (GB) Provider      Root
----           ---------     --------- --------      ----
HKCR                                   Registry      HKEY_CLASSES_ROOT
HKCU                                   Registry      HKEY_CURRENT_USER
HKLM                                   Registry      HKEY_LOCAL_MACHINE
```

Obtaining a listing of registry keys

1. Open the Windows PowerShell console.
2. Use the *Get-ChildItem* cmdlet and supply *HKLM:\ PSDrive* as the value for the *-path* argument. Specify the software key to retrieve a listing of software applications on the local machine. The resulting command is shown here:

   ```
   GCI -path HKLM:\software
   ```

   A partial listing of similar output is shown here. The corresponding keys, as displayed in Regedit.exe, are shown in Figure 3-8.

   ```
         Hive: HKEY_LOCAL_MACHINE\SOFTWARE


   Name                           Property
   ----                           --------
   ATI Technologies
   Classes
   Clients
   Intel
   Microsoft
   ODBC
   Policies
   ```

```
RegisteredApplications          Paint                    : SOFTWARE\Microsoft\
                                Windows\CurrentVersion\Applets\Paint\Capabilities
                                Windows Search           :
                                Software\Microsoft\Windows Search\Capabilities
                                Windows Disc Image Burner :
                                Software\Microsoft\IsoBurn\Capabilities
                                Windows File Explorer    : SOFTWARE\Microsoft\
                                Windows\CurrentVersion\Explorer\Capabilities
                                Windows Photo Viewer     :
                                Software\Microsoft\Windows Photo Viewer\Capabilities
                                Wordpad                  : Software\Microsoft\
                                Windows\CurrentVersion\Applets\Wordpad\Capabilities
                                Windows Media Player     :
                                Software\Clients\Media\Windows Media
                                Player\Capabilities
                                Internet Explorer        :
                                SOFTWARE\Microsoft\Internet Explorer\Capabilities
                                Windows Address Book     :
                                Software\Clients\Contacts\Address Book\Capabilities
```

This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

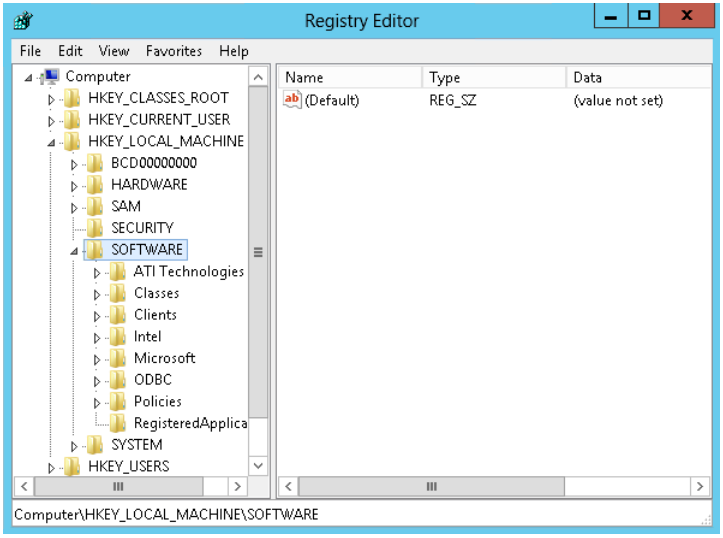

FIGURE 3-8 A Regedit.exe view of HKEY_LOCAL_MACHINE\SOFTWARE.

Searching for software

1. Use the *Get-ChildItem* cmdlet and supply a value for the *-path* argument. Use the HKLM:\ PS drive and supply a path of *SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall*. To make the command easier to read, use a single quote (') to encase the string. You can use tab completion to assist with the typing. The completed command is shown here:

```
gci -path 'HKLM:SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall'
```

   The resulting listing of software is shown in the output here, in abbreviated fashion:

```
    Hive: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall


Name                           Property
----                           --------
AddressBook
CNXT_AUDIO_HDA                 DisplayName     : Conexant 20672 SmartAudio HD
                               DisplayVersion  : 8.32.23.2
                               VersionMajor    : 8
                               VersionMinor    : 0
                               Publisher       : Conexant
                               DisplayIcon     : C:\Program
                               Files\CONEXANT\CNXT_AUDIO_HDA\UIU64a.exe
                               UninstallString : C:\Program
                               Files\CONEXANT\CNXT_AUDIO_HDA\UIU64a.exe -U -G
                                                 -Ichdrt.inf
Connection Manager             SystemComponent : 1
DirectDrawEx
DXM_Runtime
Fontcore
IE40
IE4Data
IE5BAKEX
IEData
MobileOptionPack
MPlayer2
Office15.PROPLUS               Publisher               : Microsoft Corporation
                               CacheLocation           : C:\MSOCache\All Users
                               DisplayIcon             : C:\Program Files\Common
```

2. To retrieve information on a single software package, you will need to add a *Where-Object* cmdlet. You can do this by using the up arrow key to retrieve the previous *gci -path 'HKLM:SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall'* command and pipelining the resulting object into the *Where-Object* cmdlet. Supply a value for the *name* property, as shown in the code listed here. Alternatively, supply a name from the previous output.

```
PS C:\> gci -path 'HKLM:SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall' | where
name -match 'office'
```

   This concludes this procedure.

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—all three processes use the *New-Item* cmdlet. In addition to using the *New-Item* cmdlet, you might use the *Test-Path* cmdlet to determine if the registry key already exists. You may also wish to change your working location to one of the registry drives. If you do this, you might use the *Push-Location* cmdlet, *Set-Location* and the *Pop-Location* cmdlets. This is, of course, the long way of doing things. These steps appear next.

Note 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 Microsoft TechNet article KB322756. For general information about working with the registry, see Microsoft TechNet article KB310516.

1. Store the current working location by using the *Push-Location* cmdlet.
2. Change the current working location to the appropriate registry drive by using the *Set-Location* cmdlet.
3. Use the *Test-Path* cmdlet to determine if the registry key already exists.
4. Use the *New-Item* cmdlet to create the new registry key.
5. Use the *Pop-Location* cmdlet to return to the starting working location.

The following example creates a new registry key named HSG off the HKEY_CURRENT_USERS software registry hive. It illustrates each of the five steps detailed previously.

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

The commands and the associated output from the commands appear in Figure 3-9.

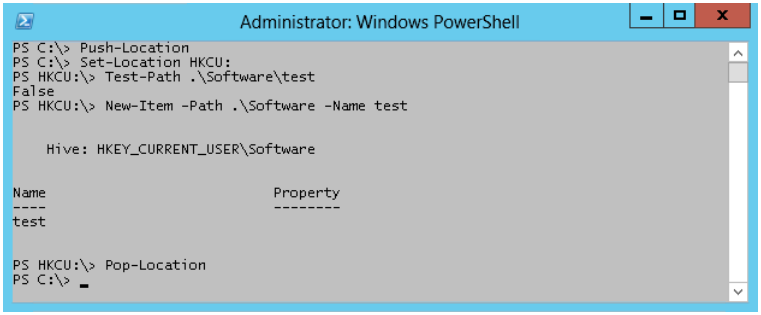

FIGURE 3-9 Creating a new registry key by using the *New-Item* cmdlet.

The short way to create a new registry key

It is not always necessary to change the working location to a registry drive when creating 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 is generated. If you want to overwrite the registry key, use the *-force* parameter. This technique works for all the Windows PowerShell providers, not just for the registry provider.

Note How to deal with an already existing registry key is one of those *design decisions* that confront IT professionals who venture far into the world of scripting. Software developers are very familiar with these types of decisions and usually deal with them in the analyzing-requirements portion of the development life cycle. IT professionals who open the Windows PowerShell ISE first and think about the design requirements second can become easily stymied, and possibly write in problems. For more information about this, see my book *Windows PowerShell 2.0 Best Practices* (Microsoft Press, 2010).

The following example creates a new registry key named *test* in the HKCU:\SOFTWARE location. Because the command includes the full path, it does not need to execute from the HKCU drive. Because the command uses the *-force* switched parameter, the command overwrites the HKCU:\SOFTWARE\TEST registry key if it already exists.

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

Note To see the *New-Item* cmdlet in action when using the *-force switched* parameter, use the *-verbose* switched parameter. The command appears here:

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

1. Include the full path to the registry key to create.
2. Use the *-force* parameter to overwrite any existing registry key of the same name.

In Figure 3-10, the first attempt to create a test registry key fails because the key already exists. The second command uses the *-force* parameter, causing the command to overwrite the existing registry key, and therefore it succeeds without creating an error.

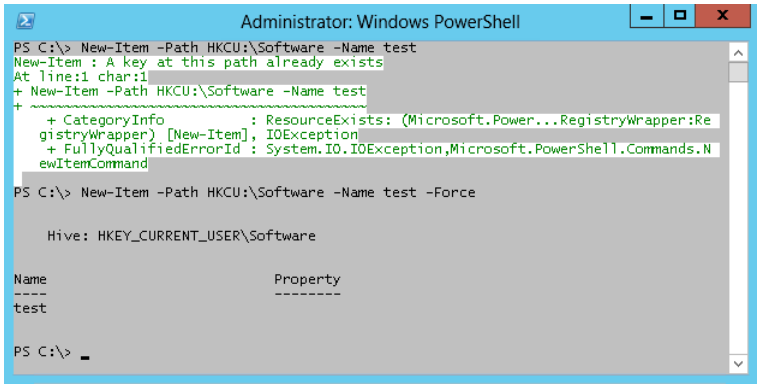

FIGURE 3-10 Use the *-force* parameter when creating a new registry key to overwrite the key if it already exists.

Setting the default value for the key

The previous 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), you can use the *Set-Item* cmdlet to assign a default value to the registry key. The steps to accomplish this appear here:

1. Use the *Set-Item* cmdlet and supply the complete path to the existing registry key.
2. Supply the default value in the *value* parameter of the *Set-Item* cmdlet.

   The following command assigns the value *test key* to the default property value of the HSG registry key contained in the HKCU:\SOFTWARE location:

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

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

It is not necessary to use the *New-Item* cmdlet to create a registry key and then to use the *Set-Item* cmdlet to assign a default value. You can combine these steps into a single command. The following command creates a new registry key with the name of HSG1 and assigns a default value of *default value* to the registry key:

```
New-Item -Path HKCU:\Software\hsg1 -Value "default value"
```

Modifying the value of a registry property value

Modifying the value of a registry property value requires using the *Set-PropertyItem* cmdlet.

1. Use the *Push-Location* cmdlet to save the current working location.
2. Use the *Set-Location* cmdlet to change to the appropriate registry drive.
3. Use the *Set-ItemProperty* cmdlet to assign a new value to the registry property.
4. Use the *Pop-Location* cmdlet to return to the original working location.

When you know that a registry property value exists, the solution is simple: you use the *Set-ItemProperty* cmdlet and assign a new value. The code that follows saves the current working location, changes the new working location to the 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.

Note The code that follows relies upon positional parameters for the *Set-ItemProperty* cmdlet. The first parameter is *-path*. Because the *Set-Location* cmdlet set the working location to the 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 would thus be *Set-ItemProperty -Path . -name newproperty -value mynewvalue*. The quotation marks in the following code are not required, but do not harm anything either.

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

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

The short way to change a registry property value

To change a registry property value simply, use the *Set-ItemProperty* cmdlet to assign a new value. Ensure you specify the complete path to the registry key. Here is an example of using 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

If you need to set a registry property value, you can set the value of that 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, as follows:

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

To determine if a registry key exists, you can simply use the *Test-Path* cmdlet. It returns *true* if the key exists and *false* if it does not exist. This technique appears here:

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

Unfortunately, this technique does not work for a registry key property. It always returns *false*—even if the registry property exists. This appears here:

```
PS C:\> Test-Path HKCU:\Software\test\newproperty
False
PS C:\> Test-Path HKCU:\Software\test\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—and using the *Test-Path* cmdlet does not work. The following procedure shows how to handle this.

Testing for a registry key property prior to writing a new value

1. 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 associated with *silentlycontinue*).
2. In the script block for the *if* statement, display a message that the registry property exists, or simply exit.
3. In the *else* statement, call *Set-ItemProperty* to create and set the value of the registry key property.

   This technique appears here:

```
if((Get-ItemProperty -Path HKCU:\Software\test -Name bogus -ea 0).bogus)
{'Propertyalready exists'}
 ELSE { Set-ItemProperty -Path HKCU:\Software\test -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. The commands used in the procedure are in the UnderstandingTheVariableProvider.txt and WorkingWithVariables.txt files. To return only cmdlets, you use the *Where-Object* cmdlet and filter on the category that is equal to cmdlet. This command is shown here:

```
Get-Help *variable | Where-Object category -eq “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 you are interested in: *name*, *category*, and *synopsis*. The revised command is shown here:

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

Note You will not get this output from Windows PowerShell 3.0 if you have not run the *Update-Help* cmdlet.

The resulting output is much easier to read and understand; it is shown here:

```
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.
```

Working with variables

1. Open the Windows PowerShell console.
2. Use the *Set-Location* cmdlet to set the working location to the Variable PS drive. Use the *sl* alias to reduce typing needs. This command is shown here:

   ```
   SL variable:\
   ```

3. Produce a complete listing of all the variables currently defined in Windows PowerShell. To do this, use the *Get-ChildItem* cmdlet. You can use the alias *gci* to produce this list. The command is shown here:

   ```
   Get-ChildItem
   ```

4. The resulting list is jumbled. Press the up arrow key to retrieve the *Get-ChildItem* command, and pipeline the resulting object into the *Sort-Object* cmdlet. Sort on the *name* property. This command is shown here:

```
Get-ChildItem | Sort Name
```

The output from the previous command is shown here:

```
Name                           Value
----                           -----
$                              variable:
?                              True
^                              sl
args                           {}
ConfirmPreference              High
ConsoleFileName
DebugPreference                SilentlyContinue
Error                          {Failed to update Help for the module(s) 'Schedule...
ErrorActionPreference          Continue
ErrorView                      NormalView
ExecutionContext               System.Management.Automation.EngineIntrinsics
false                          False
FormatEnumerationLimit         4
HOME                           C:\Users\administrator
Host                           System.Management.Automation.Internal.Host.Interna...
input                          System.Collections.ArrayList+ArrayListEnumeratorSi...
MaximumAliasCount              4096
MaximumDriveCount              4096
MaximumErrorCount              256
MaximumFunctionCount           4096
MaximumHistoryCount            4096
MaximumVariableCount           4096
MyInvocation                   System.Management.Automation.InvocationInfo
NestedPromptLevel              0
null
OutputEncoding                 System.Text.ASCIIEncoding
PID                            3308
PROFILE                        C:\Users\administrator\Documents\WindowsPowerShell...
ProgressPreference             Continue
PSBoundParameters              {}
PSCommandPath
PSCulture                      en-US
PSDefaultParameterValues       {}
PSEmailServer
PSHOME                         C:\Windows\System32\WindowsPowerShell\v1.0
PSScriptRoot
PSSessionApplicationName       wsman
PSSessionConfigurationName     http://schemas.microsoft.com/powershell/Microsoft...
PSSessionOption                System.Management.Automation.Remoting.PSSessionOption
PSUICulture                    en-US
PSVersionTable                 {PSVersion, WSManStackVersion, SerializationVersio...
PWD                            Variable:\
```

```
ShellId                         Microsoft.PowerShell
StackTrace                      at System.Management.Automation.CommandDiscover...
true                            True
VerbosePreference               SilentlyContinue
WarningPreference               Continue
WhatIfPreference                False
```

5. Use the *Get-Variable* cmdlet to retrieve a specific variable. Use the *ShellId* variable. You can use tab completion to speed up typing. The command is shown here:

```
Get-Variable ShellId
```

6. Press the up arrow key to retrieve the previous *Get-Variable ShellId* command. Pipeline the object returned into a *Format-List* cmdlet and return all properties. This is shown here:

```
Get-Variable ShellId | Format-List *
```

   The resulting output includes the description of the variable, value, and other information shown here:

```
PSPath        : Microsoft.PowerShell.Core\Variable::shellid
PSDrive       : Variable
PSProvider    : Microsoft.PowerShell.Core\Variable
PSIsContainer : False
Name          : ShellId
Description   : The ShellID identifies the current shell.  This is used by
                #Requires.
Value         : Microsoft.PowerShell
Visibility    : Public
Module        :
ModuleName    :
Options       : Constant, AllScope
Attributes    : {}
```

7. Create a new variable called *administrator*. To do this, use the *New-Variable* cmdlet. This command is shown here:

```
New-Variable administrator
```

8. Use the *Get-Variable* cmdlet to retrieve the new *administrator* variable. This command is shown here:

```
Get-Variable administrator
```

   The resulting output is shown here. Notice that there is no value for the variable.

```
Name                           Value
----                           -----
administrator
```

9. Assign a value to the new administrator variable. To do this, use the *Set-Variable* cmdlet. Specify the *administrator* variable name, and supply your given name as the value for the variable. This command is shown here:

```
Set-Variable administrator -value mred
```

10. Press the up arrow key one time to retrieve the previous *Get-Variable administrator* command. This command is shown here:

```
Get-Variable administrator
```

The output displays both the variable name and the value associated with the variable. This is shown here:

```
Name                           Value
----                           -----
administrator                  mred
```

11. Use the *Remove-Variable* cmdlet to remove the administrator variable you previously created. This command is shown here:

```
Remove-Variable administrator
```

You could also use the *Del* alias, as follows:

```
Del variable:administrator
```

12. Press the up arrow key one time to retrieve the previous *Get-Variable administrator* command. This command is shown here:

```
Get-Variable administrator
```

The variable is deleted. The resulting output is shown here:

```
Get-Variable : Cannot find a variable with name 'administrator'.
At line:1 char:13
+ Get-Variable <<<< administrator
```

This concludes this procedure.

Exploring PowerShell providers: step-by-step exercises

In this exercise, you'll explore the use of the certificate provider in Windows PowerShell. You will navigate the certificate provider by using the same types of commands used with the file system. You will then explore the environment provider by using the same methodology.

Exploring the certificate provider

1. Open the Windows PowerShell console.
2. Obtain a listing of all the properties available for use with the *Get-ChildItem* cmdlet by piping the results into the *Get-Member* cmdlet. To filter out only the properties, pipeline the results into a *Where-Object* cmdlet and specify the *membertype* to be equal to *property*. This command is shown here:

```
Get-ChildItem |Get-Member | Where-Object {$_.membertype -eq "property"}
```

3. Set your location to the Certificate drive. To identify the Certificate drive, use the *Get-PSDrive* cmdlet. Use the *Where-Object* cmdlet and filter on names that begin with the letter *c*. This is shown here:

```
Get-PSDrive |where name -like "c*"
```

   The results of this command are shown here:

```
 Name           Used (GB)     Free (GB) Provider      Root

----           ---------     --------- --------      ----

C                 110.38         38.33 FileSystem    C:\

Cert                                   Certificate   \
```

4. Use the *Set-Location* cmdlet to change to the Certificate drive:

```
Sl cert:\
```

5. Use the *Get-ChildItem* cmdlet to produce a listing of all the certificates on the machine:

```
GCI
```

   The output from the previous command is shown here:

```
Location   : CurrentUser
StoreNames : {?, UserDS, AuthRoot, CA...}

Location   : LocalMachine
StoreNames : {?, AuthRoot, CA, AddressBook...}
```

6. The listing seems somewhat incomplete. To determine whether there are additional certificates installed on the machine, use the *Get-ChildItem* cmdlet again, but this time specify the *-recurse* argument. Modify the previous command by using the up arrow key. The command is shown here:

```
GCI -recurse
```

7. The output from the previous command seems to take a long time to run and produces hundreds of lines of output. To make the listing more readable, pipe the output to a text file, and then open the file in Notepad. The command to do this is shown here:

```
GCI -recurse >C:\a.txt;notepad.exe a.txt
```

This concludes this step-by-step exercise.

In the following exercise, you'll work with the Windows PowerShell environment provider.

Examining the environment provider

1. Open the Windows PowerShell console.
2. Use the *New-PSDrive* cmdlet to create a drive mapping to the alias provider. The name of the new PS drive will be *al*. The *-PSProvider* parameter is *alias*, and the root will be dot (.). This command is shown here:

```
New-PSDrive -name al -PSProvider alias -Root .
```

3. Change your working location to the new PS drive you called *al*. To do this, use the *sl* alias for the *Set-Location* cmdlet. This is shown here:

```
SL al:\
```

4. Use the *gci* alias for the *Get-ChildItem* cmdlet, and pipeline the resulting object into the *Sort-Object* cmdlet by using the *sort* alias. Supply *name* as the property to sort on. This command is shown here:

```
GCI | Sort -property name
```

5. Press the up arrow key to retrieve the previous *gci | sort -property name* command, and modify it to use a *Where-Object* cmdlet to return aliases only when the name begins with a letter after *t* in the alphabet. Use the *where* alias to avoid typing the entire name of the cmdlet. The resulting command is shown here:

```
GCI | sort -property name | Where Name -gt "t"
```

6. Change your location back to drive C. To do this, use the *sl* alias and supply the C:\ argument. This is shown here:

```
SL C:\
```

7. Remove the PS drive mapping for al. To do this, use the *Remove-PSDrive* cmdlet and supply the name of the PS drive to remove. Note that this command does not take a trailing colon (:) or colon with backslash (:\). The command is shown here:

```
Remove-PSDrive al
```

8. Use the *Get-PSDrive* cmdlet to ensure the al drive has been removed. This is shown here:

```
Get-PSDrive
```

9. Use the *Get-Item* cmdlet to obtain a listing of all the environment variables. Use the *-path* argument and supply *env:*\ as the value. This is shown here:

```
Get-Item -path env:\
```

10. Press the up arrow key to retrieve the previous command and pipeline the resulting object into the *Get-Member* cmdlet. This is shown here:

```
Get-Item -path env:\ | Get-Member
```

The results from the previous command are shown here:

```
TypeName: System.Collections.Generic.Dictionary'2+ValueCollection[[System.
String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e0
89],[System.Collections.DictionaryEntry, mscorlib, Version=2.0.0.0, Culture=
neutral, PublicKeyToken=b77a5c561934e089]]

Name          MemberType   Definition
----          ----------   ----------
CopyTo        Method       System.Void CopyTo(DictionaryEntry[] array, Int32...
Equals        Method       System.Boolean Equals(Object obj)
GetEnumerator Method       System.Collections.Generic.Dictionary'2+ValueColl...
GetHashCode   Method       System.Int32 GetHashCode()
GetType       Method       System.Type GetType()
get_Count     Method       System.Int32 get_Count()
ToString      Method       System.String ToString()
PSDrive       NoteProperty System.Management.Automation.PSDriveInfo PSDrive=Env
PSIsContainer NoteProperty System.Boolean PSIsContainer=True
PSPath        NoteProperty System.String PSPath=Microsoft.PowerShell.Core\En...
PSProvider    NoteProperty System.Management.Automation.ProviderInfo PSProvi...
Count         Property     System.Int32 Count {get;}
```

11. Press the up arrow key twice to return to the *Get-Item -path env:*\ command. Use the Home key to move your insertion point to the beginning of the line. Add a variable called *$objEnv* and use it to hold the object returned by the *Get-Item -path env:*\ command. The completed command is shown here:

```
$objEnv=Get-Item -path env:\
```

12. From the listing of members of the environment object, find the count property. Use this property to print out the total number of environment variables. As you type **$o**, try to use tab completion to avoid typing. Also try to use tab completion as you type the *c* in *count*. The completed command is shown here:

```
$objEnv.Count
```

13. Examine the methods of the object returned by *Get-Item -path env:*\. Notice there is a *Get_Count* method. Let's use that method. The code is shown here:

```
$objEnv.Get_count
```

When this code is executed, however, the results define the method rather than execute the *Get_Count* method. These results are shown here:

```
MemberType          : Method
OverloadDefinitions : {System.Int32 get_Count()}
TypeNameOfValue     : System.Management.Automation.PSMethod
Value               : System.Int32 get_Count()
Name                : get_Count
IsInstance          : True
```

14. To retrieve the actual number of environment variables, you need to use empty parentheses at the end of the method. This is shown here:

```
$objEnv.Get_count()
```

15. If you want to know exactly what type of object is contained in the *$objEnv* variable, you can use the *GetType* method, as shown here:

```
$objEnv.GetType()
```

This command returns the results shown here:

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
False    True     ValueCollection                          System.Object
```

This concludes this exercise.

Chapter 3 quick reference

To	Do this
Produce a listing of all variables defined in a Windows PowerShell session	Use the *Set-Location* cmdlet to change location to the Variable PS drive, and then use the *Get-ChildItem* cmdlet.
Obtain a listing of all the aliases	Use the *Set-Location* cmdlet to change location to the Alias PS drive, and then use the *Get-ChildItem* cmdlet to produce a listing of aliases. Pipeline the resulting object into the *Where-Object* cmdlet and filter on the *name* property for the appropriate value.
Delete a directory that is empty	Use the *Remove-Item* cmdlet and supply the name of the directory.
Delete a directory that contains other items	Use the *Remove-Item* cmdlet and supply the name of the directory and specify the *-recurse* argument.
Create a new text file	Use the *New-Item* cmdlet and specify the *-path* argument for the directory location. Supply the *-name* argument and specify the *-type* argument as *file*. Example: *New-Item -path C:\Mytest -name Myfile.txt -type file*.
Obtain a listing of registry keys from a registry hive	Use the *Get-ChildItem* cmdlet and specify the appropriate PS drive name for the *-path* argument. Complete the path with the appropriate registry path. Example: *gci -path HKLM:\software*
Obtain a listing of all functions on the system	Use the *Get-ChildItem* cmdlet and supply the PS drive name of *function:* to the *-path* argument. Example: *gci -path function:*

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X