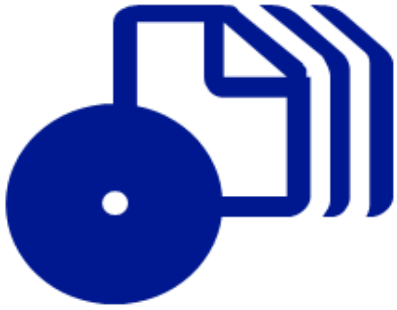
Microsoft

Windows PowerShell™ 2.0

Ed Wilson with the
Windows PowerShell
Teams at Microsoft

Best Practices

How to access your CD files



The print edition of this book includes a CD. To access the CD files, go to http://aka.ms/626461/files, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2009938599

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 WCT 4 3 2 1 0 9

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to msinput@microsoft.com.

Acquisitions Editor: Martin DelRe
Developmental Editor: Karen Szall
Project Editor: Melissa von Tschudi-Sutton
Editorial Production: Custom Editorial Productions, Inc.
Technical Reviewer: Randall Galloway; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X16-38608

This book is dedicated to Teresa Wilson, my best friend, life companion, inspiration, and wife. You made me so happy when you said yes.

Contents at a Glance

Contents

PART IV TESTING AND DEPLOYING

Acknowledgments

It always seems that I am sideswiped when working on a book project. At the outset of a project, I think, "I know this subject pretty well. This should be easy." Then I begin the outline and am confronted with my ignorance of the subject. I have worked on this book for more than 14 months—the longest of any of my eight books. Luckily, I have been aided in the project by nearly one hundred people, each of whom has expressed enthusiastic support for a project that they all acknowledged was both vitally needed and critically important.

The outline: At least once a month, I receive an e-mail, phone call, instant message, comment on Facebook, or tweet that asks me how I go about writing a book. I cannot even call my own brother without discussing the book-writing process: he is finishing his first book (on baseball). So what does this have to do with the outline? Well, I always begin with an outline. This particular outline took me more than one month to write. I had significant help in the development of the outline from Jeffrey Snover, James Brundage, Marco Shaw, Bill Mell, Jit Banerjie, Chris Bellée, and Pete Christensen.

The sidebars: One of the cool things about a Best Practices book is the sidebars. Sidebars are short text pieces that are written by well-known industry experts as well as members of the Windows PowerShell product group. I have been extremely fortunate to have developers, test engineers, architects, field engineers, network administrators, and consultants contribute sidebars for this book. They wrote because they believed in the value of creating a book on best practices and not because it was their job or because they were otherwise compensated. These contributors are current and former employees of Microsoft. Some are Microsoft MVPs, and others will probably become Microsoft MVPs in the future. Some are moderators for the Official Scripting Guys forum, and others are moderators for other scripting forums. I even rounded up a few former Microsoft Scripting Guys. In every case, my colleagues were happy to supply a sidebar. I enjoyed reading the sidebars as much as I am sure you will enjoy them.

The following people provided assistance, suggestions, and/or contributions to sidebars for the book: Alexander Riedel, Andrew Willett, Andy Schneider, Ben Pearce, Bill Mell, Bill Stewart, Brandon Shell, Chris Bellée, Clint Huffman, Dan Harman, Daniele Muscetta, Dave Schwinn, Dean Tsaltas, Don Jones, Enrique Cedeno, Georges Maheu, Hal Rottenberg, James Brundage, James Craig Burley, James Turner, Jeffrey Hicks, Jeffrey Snover, Juan Carlos Ruiz Lopez, Keith Hill, Lee Holmes, Luís Canastreiro, Osama Sajid, Peter Costantini, Rahul Verma, Richard Norman, Richard Siddaway, Vasily Gusev, Alex K. Angelopoulos, David Zazzo,

Glenn Seagraves, Joel Bennett, Marco Shaw, Oisín Grehan, Pete Christensen, and Thomas Lee.

Special mention goes to Melissa von Tschudi-Sutton for all of her patience and guidance. During the writing of this book, I went from traveling every week (with a significant amount of international travel to Australia, Canada, Germany, Denmark, Portugal, and Mexico) to my current position as the Microsoft Scripting Guy. This change caused some disruption to my work schedule. Additionally, this project also saw a nearly two-month hiatus due to a surgery-inducing accident that I suffered in my woodworking shop. Throughout all of these changes, Melissa was a real trouper, and I could not have asked for a more understanding editor. She was awesome!

Randall Galloway, who is a Microsoft Technical Account manager and my technical editor on this book, was great. He went beyond the call of duty by even responding to style and grammar questions posed by Jan Clavey, who was my copy editor. Rose Marie Kuebbing and Megan Smith-Creed, who were project managers for this book, also contributed in significant ways to keeping the book on schedule. As the content development manager, Karen Szall did a wonderful job of helping me quickly come up to speed on all of the new features of a Best Practices book. I extend a hearty "well done" to all of you!

Windows PowerShell 2.0 Best Practices would not exist without the tireless efforts of my agent and friend, Claudette Moore, of the Moore Literary Agency. Claudette is amazing! That she can navigate all of the intricacies of a book contract and negotiate with editors and publishers while still keeping her sanity is truly an art form. Martin DelRe, my acquisitions editor from Microsoft Press, is a staunch supporter of scripting technology and makes me feel as if my book is the most important book to be published this year. His personal attention to the project is truly appreciated.

My biggest acknowledgment, however, must go to Teresa Wilson, who is my number one editor, critic, and fan. Nothing I write—whether an e-mail to John Merril (my manager at Microsoft), a Hey Scripting Guy! article, or a book for Microsoft Press—leaves our house without her careful scrutiny. She has read every word of every book that I have ever written. Incidentally, she is an accounting type of person and not a script guru or a wordsmith, even though she is sometimes referred to as "the scripting wife" in some of my Hey Scripting Guy! articles.

Even with all of this help from my friends and colleagues, I am absolutely certain that there will be a few errors in this book. All of those belong to me.

Introduction

One of the great things about being the Microsoft Scripting Guy is the interaction I have with customers—people who are using Windows PowerShell to manage enterprise networks of every size and description: pure Microsoft networks, heterogeneous networks, and even networks running software that is obsolete and no longer supported. Within all of these interactions with customers runs a common thread: "How can I more efficiently manage my network?"

Windows PowerShell is the Microsoft answer to the question of more effective and efficient network management. At TechEd 2009 in Los Angeles, the Windows PowerShell sessions were the most heavily attended of all of the sessions offered. While I was manning the Microsoft Scripting Guys booth, nearly all of the questions I received were about Windows PowerShell. My Friday afternoon talk on using Windows PowerShell to manage the Windows 7 desktop saw the highest attendance of any session offered during that time slot. Quite frankly, I was surprised that anyone attended because the convention center was a scant 15 miles from Disneyland. I felt like I was competing with Mickey Mouse for the attendees' attention after a week of sessions.

Why I Wrote This Book

"Windows administrators do not script." I have heard this truism for years. Prior to becoming the Microsoft Scripting Guy, I delivered workshops to Microsoft Premier Customers around the world on VBScript, Windows Management Instrumentation (WMI), and Windows PowerShell. Because these workshops were so popular, I had to train other instructors to assist with the demand for training. These scripting workshops were the highest-rated workshops offered through Microsoft Premier Services. There is definitely a demand and an interest in scripting. Over the years, the questions we fielded began to repeat themselves: "What is the best way to do this?" or "What is the best way to do that?" The scripting instructors formed a work group as we researched answers to these common questions. At one point, we discussed the creation of a scripting Best Practices workshop, and much of that work has been incorporated into this book.

On Form and Function

To a very large extent, the use of your script will dictate the form that it takes. Will all of the best practices in this book apply to every script that you write? No. The goal of administrative scripting is to do work, and anything that enables you to

efficiently accomplish your task is probably acceptable. However, there is a point when a quickly hobbled-together script can be more trouble than it is worth, particularly if it introduces security concerns or reliability issues.

Several benefits can be derived from following best practices. The first is that a well-crafted script will be easier to understand. If it is easier to read and understand, it will be easier to modify when the time comes to add capability to the script. If a problem arises with the script, it will also be easier to troubleshoot.

For Whom Is This Book?

Windows PowerShell 2.0 Best Practices is aimed at experienced IT professionals including IT architects, engineers, administrators, and support professionals who are working in large organizations and in the enterprise. IT pros who work in smaller companies would also benefit from most of the guidelines concerning the structuring of code, many of the tips and tricks, and much of the "how to" sections regarding common everyday problems. The audience is expected to have obtained the level of Windows PowerShell understanding presented in *Microsoft Windows PowerShell Step by Step* (Microsoft Press, 2007). The audience is assumed to know and understand the basics of Windows PowerShell: fundamental looping constructs, basic decision-making code blocks, and a rudimentary understanding of the use of cmdlets, which is the type of information they would obtain in a three-day class. In this book, there are two level-setting chapters as well as references back to topics covered in more detail in *Microsoft Windows PowerShell Step by Step*. This book is a 300-level book, but the planning and managing section will be very useful to IT managers seeking guidance on establishing scripting procedures for an enterprise organization.

How Is This Book Organized?

This book is organized into five sections as listed here:

- Part I: Introduction
- Part II: Planning
- Part III: Designing
- Part IV: Testing and Deploying
- Part V: Optimizing

The first four chapters introduce many of the new features of Windows PowerShell 2.0. In Chapter 1, I cover a variety of the new cmdlets that are introduced in Windows PowerShell 2.0 as well as best practices for deploying Windows PowerShell 2.0 to different portions of the network. In Chapter 2, I discuss the new

WMI features in Windows PowerShell 2.0 and provide tips for incorporating these features into a management strategy for your network. Chapters 3 and 4 cover Active Directory.

I begin the planning section of the book in Chapter 5 by identifying scripting opportunities. Here I talk about the different types of technology available to the scripter. In Chapters 6 and 7, I present best practices for configuring the scripting environment and avoiding scripting pitfalls. Chapter 8 concludes the planning section by discussing best practices for engendering scripting collaboration within the corporate IT environment.

In the section about design, I go into a great amount of detail about functions and help. There are significant new features in Windows PowerShell 2.0 that will ratchet your enterprise scripts to a new level of functionality. Chapter 11 covers modules, which are another new Windows PowerShell 2.0 feature that provides the ability to store, share, and deploy functions and other program elements in an easy-to-reuse manner. Because there are so many methods to get data into and out of a Windows PowerShell script, I provide some guidelines to help you determine the best methodology for your particular application. You cannot always predict what the environment will be when your script is run. Therefore, in Chapter 13, I discuss best practices for handling errors.

Of course, many errors can be avoided if a script is thoroughly tested prior to deployment, so I've included a section on testing and deploying. Script testing is discussed in Chapter 14, including the types of tests to run and the type of environment to use for your script testing. In Chapter 15, I discuss best practices for running scripts. Here I talk about working with the script execution policy, code signing, and other topics that are often confusing.

In the final section of the book, I cover optimization. In Chapter 16, I present different options for implementing logging in your script. I conclude *Windows PowerShell 2.0 Best Practices* with Chapter 17, where I cover the new troubleshooting and debugging tools included in Windows PowerShell 2.0. There are many choices for you to use, and I provide quite a bit of guidance, tips, and best practices to effectively troubleshoot a Windows PowerShell script.

NOTE **Sidebars are provided by individuals in the industry as examples for informational purposes only and may not represent the views of their employers. No warranties, express, implied, or statutory, are made as to the information provided in sidebars.**

What This Book Is Not

In *Windows PowerShell 2.0 Best Practices*, I assume that you have a basic understanding of Windows PowerShell. While I go to great lengths to explain the new features of Windows PowerShell 2.0, I do not always devote the same detail to the features of Windows PowerShell 1.0. Even though I think you can learn to use Windows PowerShell from reading this book, it is not a comprehensive text, nor is it a tutorial or even a Windows PowerShell reference book. The organization of the book, while appropriate for dealing with Windows PowerShell best practices, is not optimal for learning Windows PowerShell from scratch. I therefore recommend the following books as either prereading prior to approaching this book or, at a minimum, as supplemental reading while you are perusing this book:

- *Microsoft Windows PowerShell Step by Step* (Microsoft Press, 2007)
- *Windows PowerShell Scripting Guide* (Microsoft Press, 2008)

DIGITAL CONTENT FOR DIGITAL BOOK READERS **If you bought a digital-only edition of this book, you can enjoy select content from the print edition's companion media. Visit *http://www.microsoftpressstore.com/title/9780735626461* to get your downloadable content. This content is always up to date and available to all readers.**

System Requirements

This book is designed to be used with the following software:

- Windows XP or later
- 512 MB of RAM
- P4 processor or higher
- 100 MB of available disk space
- Internet Explorer 6.0 or later
- Windows PowerShell 2.0 or later

The following list details the minimum system requirements needed to run the companion media provided with this book:

- Windows XP, with the latest service pack installed and the latest updates from Microsoft Update Service
- CD-ROM drive
- Display monitor capable of 1024 × 768 resolution

- Microsoft Mouse or compatible pointing device
- Adobe Reader for viewing the eBook (Adobe Reader is available as a download at *http://www.adobe.com*.)

Support for This Book

Every effort has been made to ensure the accuracy of this book. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article accessible via the Microsoft Help and Support site. Microsoft Press provides support for books, including instructions for finding Knowledge Base articles, at the following Web site: *http://www.microsoft.com/learning/support/books/*.

If you have questions regarding the book that are not answered by visiting the site above or viewing a Knowledge Base article, send them to Microsoft Press via e-mail to *mspinput@microsoft.com*. Please note that Microsoft software product support is not offered through these addresses.

We Want to Hear from You

We welcome your feedback about this book. Please share your comments and ideas via the following short survey: *http://www.microsoft.com/learning/booksurvey*. Your participation will help Microsoft Press create books that better meet your needs and your standards.

NOTE **We hope that you will give us detailed feedback via our survey. If you have questions about our publishing program, upcoming titles, or Microsoft Press in general, we encourage you to interact with us via Twitter at *http://twitter.com/MicrosoftPress*. For support issues, use only the e-mail address shown above.**

About the Companion Media

On this book's companion media, I include all of the scripts that are discussed in the book. The scripts are stored in folders that correspond to the chapters in the book. There is also a folder named Extras that contains extra scripts that I wrote while working on the chapters.

Writing extra scripts has become a tradition for me throughout all of the five scripting books that I have written. Some of these are simply modifications of the scripts in the chapters, while others are scripts that I wrote for fun that I wanted to share. You will also find some planning templates and other tools that might be useful to you as you develop your corporate scripting standards. Last but certainly not least, you will find a document that contains a summary of the best practices from this book. It is my sincere hope that this document will become a quick reference that you can use to refer to the best practices for scripting when you have a question while writing a script.

CHAPTER 7

Avoiding Scripting Pitfalls

Knowing what you should not script is as important as knowing what you should script. There are times when creating a Windows PowerShell script is not the best approach to a problem due to the lack of support in a particular technology or to project complexity. In this chapter, you will be introduced to some of the red flags that signal danger for a potential script project.

Lack of cmdlet Support

It is no secret that cmdlet support is what makes working with Windows PowerShell so easy. If you need to check the status of the bits service, the easiest method is to use the `Get-Service` cmdlet as shown here.

```
Get-Service –name bits
```

To find information about the explorer process, you can use the `Get-Process` cmdlet as shown here.

```
Get-Process -Name explorer
```

If you need to stop a process, you can easily use the `Stop-Process` cmdlet as shown here.

```
Stop-Process -Name notepad
```

You can even check the status of services on a remote computer by using the *–computername* switch from the `Get-Service` cmdlet as shown here.

```
Get-Service -Name bits -ComputerName vista
```

> ***IMPORTANT*** **If you are working in a cross-domain scenario in which authentication is required, you will not be able to use `Get-Service` or `Get-Process` because those cmdlets do not have a *–credential* parameter. You need to use one of the remoting cmdlets, such as `Invoke-Command`, which allows you to supply an authentication context.**

You can check the BIOS information on a local computer and save the information to a comma-separated value file with just a few lines of code. An example of such a script is the ExportBiosToCsv.ps1 script.

ExportBiosToCsv.ps1

```
$path = "c:\fso\bios.csv"
Get-WmiObject -Class win32_bios |
Select-Object -property name, version |
Export-CSV -path $path –noTypeInformation
```

Without cmdlet support for selecting objects and exporting them to a CSV file format, you might be tempted to use *filesystemobject* from Microsoft VBScript fame. If you take that approach, the script will be twice as long and not nearly as readable. An example of a script using *filesystemobject* is the FSOBiosToCsv.ps1 script.

FSOBiosToCsv.ps1

```
$path = "c:\fso\bios1.csv"
$bios = Get-WmiObject -Class win32_bios
$csv = "Name,Version`r`n"
$csv +=$bios.name + "," + $bios.version
$fso = new-object -comobject scripting.filesystemobject
$file = $fso.CreateTextFile($path,$true)
$file.write($csv)
$file.close()
```

Clearly, the ability to use built-in cmdlets is a major strength of Windows PowerShell. One problem with Windows Server 2008 R2 and Windows PowerShell 2.0 is the number of cmdlets that exist, which is similar to the problem experienced by Windows Exchange Server administrators. Because there are so many cmdlets, it is difficult to know where to begin. A quick perusal of the Microsoft Exchange Team's blog and some of the Exchange forums reveals that the problem is not writing scripts, but finding the one cmdlet of the hundreds of possible candidates that performs the specific task at hand. If you factor in community-developed cmdlets and third-party software company cmdlet offerings, you have a potential environment that encompasses thousands of cmdlets.

Luckily, the Windows PowerShell team has a plan to address this situation—standard naming conventions. The `Get-Help`, `Get-Command`, and `Get-Member` cmdlets were discussed in Chapter 1, "Assessing the Scripting Environment," but they merit mention here. If you are unaware of a specific cmdlet feature or even the existence of a cmdlet, you are forced to implement a workaround that causes additional work or that might mask hidden mistakes. Given the choice between a prebaked cmdlet and a create-your-own solution, the prebaked cmdlet should be used in almost all cases. Therefore, instead of assuming that a cmdlet or feature does not exist, you should spend time using `Get-Help`, `Get-Command`, and `Get-Member` before embarking on a lengthy development effort. In this chapter, you will examine some of the potential pitfalls that can develop when you do not use cmdlets.

Complicated Constructors

If you do not have support from cmdlets when developing an idea for a script, this indicates that there may be a better way to do something and should cause you to at least consider your alternatives.

In the GetRandomObject.ps1 script, a function named *GetRandomObject* is created. This function takes two input parameters: one named *$in* that holds an array of objects and the other named *$count* that controls how many items are randomly selected from the input object.

The `New-Object` cmdlet is used to create an instance of the *System.Random* Microsoft .NET Framework class. The new instance of the class is created by using the default constructor (no seed value supplied) and is stored in the *$rnd* variable.

A *for . . . next* loop is used to loop through the collection—once for each selection desired. The next method of the *System.Random* class is used to choose a random number that resides between the number 1 and the maximum number of items in the input object. The random number is used to locate an item in the array by using the index so that the selection of the item from the input object can take place. The GetRandomObject.ps1 script is shown here.

GetRandomObject.ps1

```
Function GetRandomObject($in,$count)
{
 $rnd = New-Object system.random
 for($i = 1 ; $i -le $count; $i ++)
 {
  $in[$rnd.Next(1,$a.length)]
 } #end for
} #end GetRandomObject

# *** entry point ***
$a = 1,2,3,4,5,6,7,8,9
$count = 3
GetRandomObject -in $a -count $count
```

While there is nothing inherently wrong with the GetRandomObject.ps1 script, you can use the `Get-Random` cmdlet when working with Windows PowerShell 2.0 to accomplish essentially the same objective as shown here.

```
$a = 1,2,3,4,5,6,7,8,9
Get-Random -InputObject $a -Count 3
```

Clearly, by using the native `Get-Random` cmdlet, you can save yourself a great deal of time and trouble. The only reason to use the GetRandomObject.ps1 script is that it works with both Windows PowerShell 1.0 and PowerShell 2.0.

One advantage of using a cmdlet is that you can trust it will be implemented correctly. At times, .NET Framework classes have rather complicated constructors that are used to govern the way the instance of a class is created. A mistake that is made when passing a value for one of these constructors does not always mean that an error is generated. It is entirely possible that the code will appear to work correctly, and it can therefore be very difficult to spot the problem.

An example of this type of error is shown in the BadGetRandomObject.ps1 script in which an integer is passed to the constructor for the *System.Random* .NET Framework class. The problem is that every time the script is run, the same random number is generated. While this particular bad implementation is rather trivial, it illustrates that the potential exists for logic errors that often require detailed knowledge of the utilized .NET Framework classes to troubleshoot.

BadGetRandomObject.ps1

```
Function GetRandomObject($in,$count,$seed)
{
 $rnd = New-Object system.random($seed)
 for($i = 1 ; $i -le $count; $i ++)
 {
  $in[$rnd.Next(1,$a.length)]
 } #end for
} #end GetRandomObject

# *** entry point ***
$a = 1,2,3,4,5,6,7,8,9
$count = 3
GetRandomObject -in $a -count $count -seed 5
```

The *System.Random* information is contained in MSDN, but it is easy to overlook some small detail because there is so much documentation and some of the classes are very complicated. When the overlooked detail does not cause a run-time error and the script appears to work properly, then you have a potentially embarrassing situation at best.

Version Compatibility Issues

While the Internet is a great source of information, it can often lead to confusion rather than clarity. When you locate a source of information, it may not be updated to include the current version of the operating system. This update situation is worsening due to a variety of complicating factors such as User Account Control (UAC), Windows Firewall, and other security factors that have so many different configuration settings that it can be unclear whether an apparent failure is due to a change in the operating system or to an actual error in the code. As an example, suppose that you decide to use the *WIN32_Volume* Windows Management Instrumentation (WMI) class to determine information about your disk drives. First, you need to realize that the WMI class does not exist on any operating system older than Microsoft Windows Server 2003; it is a bit surprising that the class does not exist on Windows XP. When you try the following command on Windows Vista, however, it generates an error.

```
Get-WmiObject -Class win32_volume -Filter "Name = 'c:\'"
```

The first suspect when dealing with Windows Vista and later versions is user rights. You open the Windows PowerShell console as an administrator and try the code again; it fails. You then wonder whether the error is caused by the differences between expanding quotes and literal quotes. After contemplation, you decide to write the filter to take advantage of literal strings. The problem is that you have to escape the quotes, which involves more work, but it is worth the effort if it works. So, you come up with the following code that, unfortunately, also fails when it is run.

```
Get-WmiObject -Class win32_volume -Filter 'Name = ''c:\'''
```

This time, you decide to actually read the error message. Here is the error that was produced by the previous command.

```
Get-WmiObject : Invalid query
At line:1 char:14
+ Get-WmiObject <<<<  -Class win32_volume -Filter "Name = 'c:\' "
    + CategoryInfo          : InvalidOperation: (:) [Get-WmiObject],
    ManagementException
    + FullyQualifiedErrorId : GetWMIManagementException,
    Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

You focus on the line that says invalid operation and decide that perhaps the backslash is a special character. When this is the problem, you need to escape the backslash; therefore, you decide to use the escape character to make one more attempt. Here is the code you create.

```
Get-WmiObject -Class win32_volume -Filter "Name = 'c:`\' "
```

Even though this is a good idea, the code still does not work and once again generates an error as shown here.

```
Get-WmiObject : Invalid query
At line:1 char:14
```

```
+ Get-WmiObject <<<<  -Class win32_volume -Filter "Name = 'c:`\' "
    + CategoryInfo          : InvalidOperation: (:) [Get-WmiObject],
    ManagementException
    + FullyQualifiedErrorId : GetWMIManagementException,
    Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

Next, you search to determine whether you have rights to run the query. (I know that you are running the console with Administrator rights, but some processes deny access even to the Administrator, so it is best to check.) The easiest way to check your rights is to perform the WMI query and omit the *–filter* parameter as shown here.

```
Get-WmiObject -Class win32_volume
```

This command runs without generating an error. You may assume that you cannot filter the WMI class at all and decide that the class is a bit weird. You may decide to write a different filter and see whether it will accept the syntax of a new filter, such as the following line of code.

```
Get-WmiObject -Class win32_volume -Filter "DriveLetter = 'c:'"
```

The previous command rewards you with an output similar to the one shown here.

```
PS C:\> Get-WmiObject -Class win32_volume -Filter "DriveLetter = 'c:'"


__GENUS                   : 2
__CLASS                   : Win32_Volume
__SUPERCLASS              : CIM_StorageVolume
__DYNASTY                 : CIM_ManagedSystemElement
__RELPATH                 : Win32_Volume.DeviceID="\\\\?\\Volume{5a4a2fe5-70
                            f0-11dd-b4ad-806e6f6e6963}\\"
__PROPERTY_COUNT          : 44
__DERIVATION              : {CIM_StorageVolume, CIM_StorageExtent, CIM_Logic
                            alDevice, CIM_LogicalElement...}
__SERVER                  : MRED1
__NAMESPACE               : root\cimv2
__PATH                    : \\MRED1\root\cimv2:Win32_Volume.DeviceID="\\\\?\
                            \Volume{5a4a2fe5-70f0-11dd-b4ad-806e6f6e6963}\\"
Access                    :
Automount                 : True
Availability              :
BlockSize                 : 4096
BootVolume                : True
Capacity                  : 158391595008
Caption                   : C:\
Compressed                : False
ConfigManagerErrorCode    :
ConfigManagerUserConfig   :
```

```
CreationClassName            :
Description                  :
DeviceID                     : \\?\Volume{5a4a2fe5-70f0-11dd-b4ad-806e6f6e6963}
                               \
DirtyBitSet                  :
DriveLetter                  : C:
DriveType                    : 3
ErrorCleared                 :
ErrorDescription             :
ErrorMethodology             :
FileSystem                   : NTFS
FreeSpace                    : 23077511168
IndexingEnabled              : True
InstallDate                  :
Label                        :
LastErrorCode                :
MaximumFileNameLength        : 255
Name                         : C:\
NumberOfBlocks               :
PageFilePresent              : False
PNPDeviceID                  :
PowerManagementCapabilities  :
PowerManagementSupported     :
Purpose                      :
QuotasEnabled                :
QuotasIncomplete             :
QuotasRebuilding             :
SerialNumber                 : 1893548344
Status                       :
StatusInfo                   :
SupportsDiskQuotas           : True
SupportsFileBasedCompression : True
SystemCreationClassName      :
SystemName                   : MRED1
SystemVolume                 : False
```

NOTE **When working with scripting, network administrators and consultants often use workarounds because our job is to "make things work." Sometimes, Scripting Guys end up using workarounds as well. After all, my job is to write a daily Hey Scripting Guy! column, which means that I have a deadline every day of the week. Concerning the previous *Win32_Volume* WMI class example, I always use the *DriveLetter* property when performing the WMI query. Years ago, after several hours of experimentation with this example, I determined that perhaps the *name* property was broken and therefore avoided using it when performing demonstrations when I was teaching classes. Luckily, no student ever asked me why I use *DriveLetter* instead of the *name* property in any of my queries!**

If you return to the error message generated by the earlier queries, the InvalidOperation CategoryInfo field might cause you to reconsider the backslash. Your earlier attempts to escape the backslash were on the right track. The problem revolves around the strange mixture of the WMI Query Language (WQL) syntax and Windows PowerShell syntax. The *–filter* parameter is definitely Windows PowerShell syntax, but you must supply a string that conforms to WQL dialect inside this parameter. This is why you use the equal sign for an operator instead of the Windows PowerShell *–eq* operator when you are inside the quotation marks of the *–filter* parameter. To escape the backslash in the WQL syntax, you must use another backslash as found in C or C++ syntax. The following code filters out the drive based on the name of the drive.

```
Get-WmiObject -Class win32_volume -Filter "Name = 'c:\\'"
```

IMPORTANT **Use of the backslash to escape another backslash is a frustrating factor when using WMI. While our documentation in MSDN is improving, we still have a way to go in this arena. Because this WMI class does not behave as you might expect, I have filed a documentation bug for the *name* property of the *Win32_Volume* class. The result will be an additional note added to the description of the property. I have since found a few more places where the backslash is used as an escape character, and I will file bugs on them as well.**

As a best practice, you can write a script to return the WMI information from the *WIN32_Volume* class and hide the escape details from the user. An example of such a script is the GetVolume.ps1 script. The script accepts two command-line parameters: *–drive* and *–computer*. The drive is supplied as a drive letter followed by a colon. By default, the script returns information from the C: drive on the local computer. In the *Get-Volume* function, the *–drive* value is concatenated with the double backslash and is then submitted to the `Get-WmiObject` cmdlet. One interesting aspect is the use of single quotes around the *$drive* variable. Remember that, inside the *–filter* parameter, the script uses WQL syntax and not Windows PowerShell syntax. The single quote is simply a single quote, and you do not need to worry about the difference between an expanding or a literal quotation. The GetVolume.ps1 script is shown here.

GetVolume.ps1

```
Param($drive = "C:", $computer = "localhost")
Function Get-Volume($drive, $computer)
{
 $drive += "\\"
 Get-WmiObject -Class Win32_Volume -computerName $computer `
 -filter "Name = '$drive'"
}

Get-Volume -Drive $drive -Computer $computer
```

If you need to work in a cross-domain situation, you need to pass credentials to the remote computer. The `Get-WmiObject` cmdlet contains the *–credential* parameter that can be used in just such a situation. Because the `Get-WmiObject` cmdlet uses WMI in the background, the problem is that you are not allowed to pass credentials for a local connection. Local WMI scripts always run in the context of the calling user—that is, the one who is actually launching the script. This means that you cannot use the *–credential* parameter for a local script to allow a nonprivileged user to run the script with administrator rights. You can use the *–credential* parameter with remote connections. In addition, you are not allowed to have the *–credential* parameter in the `Get-WmiObject` cmdlet and leave it blank or null because this also generates an error. The solution is to check whether the script is running against the local computer; if it is, use the *Get-Volume* function from the previous script. If it is working remotely, the script should use a different function that supplies the *–credential* parameter as shown in the GetVolumeWithCredentials.ps1 script.

GetVolumeWithCredentials.ps1

```
Param(
      $drive = "C:",
      $computer = "localhost",
      $credential
     )
Function Get-Volume($drive, $computer)
{
 $drive += "\\"
 Get-WmiObject -Class Win32_Volume -computerName $computer `
 -filter "Name = '$drive'"
} #end Get-Volume

Function Get-VolumeCredential($drive, $computer,$credential)
{
 $drive += "\\"
 Get-WmiObject -Class Win32_Volume -computerName $computer `
 -filter "Name = '$drive'" -credential $credential
} #end Get-VolumeCredential

# *** Entry point to script
If($computer -eq "localhost" -AND $credential)
  { "Cannot use credential for local connection" ; exit }
Elseif ($computer -ne "localhost" -AND $credential)
  {
   Get-VolumeCredential -Drive $drive -Computer $computer `
   -Credential $credential
  }
Else
 { Get-Volume -Drive $drive -Computer $computer }
```

INSIDE TRACK

Choosing the Right Script Methodology

Luis Canastreiro, Premier Field Engineer
Microsoft Corporation, Portugal

When I am writing a script, often there are many ways of accomplishing the same task. If I am writing a VBScript, for example, I prefer to use a Component Object Model (COM) object rather than shelling out and calling an external executable because COM is native to VBScript. The same principle holds when I am writing a Windows PowerShell script. I prefer to use the .NET Framework classes if a Windows PowerShell cmdlet is not available because PowerShell is built on the .NET Framework.

Of course, my number-one preference is to use a cmdlet if it is available to me because a cmdlet will hide the complexity of dealing directly with the .NET Framework. By this I mean that there are some .NET Framework classes that at first glance appear to be simple. However, when you begin to use them, you realize that they contain complicated constructors. If you are not an expert with that particular class, you can make a mistake that will not be realized until after much testing. If a cmdlet offers the required features and if it solves my problem, then the cmdlet is my first choice.

As an example, there are several ways to read and write to the registry. You can use the *regread* and *regwrite* VBScript methods, the *stdRegProv* WMI class, the .NET Framework classes, or even various command-line utilities to gain access to the registry. My favorite method of working with the registry is to use the Windows PowerShell registry provider and the various `*-item` and `*-itemproperty` cmdlets. These cmdlets are very easy to use, and I only need to open the Windows PowerShell shell to accomplish everything I need to do with these cmdlets.

When I am writing a new script, I always like to create small generic functions, which offer a number of advantages. These functions make it easy for me to test the script while I am in the process of writing it. I only need to call the function or functions on which I am working. I can leave the other code unfinished if I need to test it later. The functions are easy to reuse or to improve as time goes by. I run the script by creating a main function whose primary purpose is to initialize the environment and manage the global flow of the script by calling the appropriate functions at the proper time.

Trapping the Operating System Version

Given the differences between the various versions of the Windows operating system, it is a best practice to check the version of the operating system prior to executing the script if you know that there could be version compatibility issues. There are several different methods to check version compatibility. In Chapter 5, "Identifying Scripting Opportunities," you used the *System.Environment* .NET Framework class to check the operating system version in the GetOsVersionFunction.ps1 script. While it is true that you can use remoting to obtain information from this class remotely, you can also achieve similar results by using the *Win32_OperatingSystem* WMI class. The advantage of this approach is that WMI automatically remotes.

The GetVersion.ps1 script accepts a single command-line parameter, *$computer*, and is set by default to the localhost computer. The entry point to the script passes the value of the *$computer* variable and a reference type *$osv* variable to the *Get-OSVersion* function. Inside the *Get-OSVersion* function, the `Get-WMIObject` cmdlet is first used to query the *Win32_OperatingSystem* WMI class from the computer that is targeted by the *$computer* variable. The resulting management object is stored in the *$os* variable.

The *Switch* statement is used to evaluate the *version* property of the *Win32_OperatingSystem* class. If the value of the *version* property is equal to 5.1.2600, the *Value* property of the *$osv* reference type variable is set equal to "xp". This type of logic is repeated for the value 5.1.3790, which is the build number for the Windows 2003 server.

A problem arises if the version number is 6.0.6001 because both Windows Vista and Windows Server 2008 have the same build number. This is why the script stores the entire *Win32_OperatingSystem* management object in the *$os* variable instead of retrieving only the version attribute. The *ProductType* property can be used to distinguish between a workstation and a server. The possible values for the *ProductType* property are shown in Table 7-1.

TABLE 7-1 *Win32_OperatingSystem ProductType* Values and Associated Meanings

VALUE	MEANING
1	Workstation
2	Domain controller
3	Server

Once the version of the operating system is detected, then a single word or number representing the operating system is assigned to the *Value* property of the reference variable. In the GetVersion.ps1 script, this value is displayed at the console. The complete GetVersion.ps1 script is shown here.

GetVersion.ps1

```
Param($computer = "localhost")

Function Get-OSVersion($computer,[ref]$osv)
{
 $os = Get-WmiObject -class Win32_OperatingSystem `
       -computerName $computer
 Switch ($os.Version)
  {
    "5.1.2600" { $osv.value = "xp" }
    "5.1.3790" { $osv.value = "2003" }
    "6.0.6001"
                {
                  If($os.ProductType -eq 1)
                    {
                     $osv.value = "Vista"
                    } #end if
                  Else
                    {
                     $osv.value = "2008"
                    } #end else
                } #end 6001
     DEFAULT { "Version not listed" }
  } #end switch
} #end Get-OSVersion

# *** entry point to script ***
$osv = $null
Get-OSVersion -computer $computer -osv ([ref]$osv)
$osv
```

The GetVersion.ps1 script returns a single word to indicate the version of the operating system. You can use this script from the command line to quickly check the operating system version as shown here.

```
PS C:\bp> .\GetVersion.ps1
Vista
PS C:\bp> .\GetVersion.ps1 -c berlin
2008
PS C:\bp> .\GetVersion.ps1 -c lisbon
xp
```

The GetVersion.ps1 script is written as a function to permit easy inclusion into other scripts, which allows you to perform the operating system version check and then decide whether you want to continue processing the script. An example of this approach is shown in the GetVersionGetVolume.ps1 script.

GetVersionGetVolume.ps1

```
Param($drive = "C:", $computer = "localhost")

Function Get-OSVersion($computer,[ref]$osv)
{
 $os = Get-WmiObject -class Win32_OperatingSystem `
           -computerName $computer
Switch ($os.Version)
  {
    "5.1.2600" { $osv.value = "xp" }
    "5.1.3790" { $osv.value = "2003" }
    "6.0.6001"
                     {
                       If($os.ProductType -eq 1)
                          {
                            $osv.value = "Vista"
                          } #end if
                       Else
                         {
                           $osv.value = "2008"
                          } #end else
                     } #end 6001
     DEFAULT { "Version not listed" }
  } #end switch
} #end Get-OSVersion

Function Get-Volume($drive, $computer)
{
 $drive += "\\"
 Get-WmiObject -Class Win32_Volume -computerName $computer `
 -filter "Name = '$drive'"
} #end Get-Volume

# *** entry point to script ***
$osv = $null
Get-OSVersion -computer $computer -osv ([ref]$osv)
if($osv -eq "xp") { "Script does not run on XP" ; exit }
Get-Volume -Drive $drive -Computer $computer
```

Lack of WMI Support

Windows Management Instrumentation has been in existence since the days of Microsoft Windows NT 4.0. In the years since its introduction, every new version of Windows has added WMI classes and, at times, additional methods to existing WMI classes. One advantage of WMI is its relatively consistent approach to working with software and hardware. Another advantage of WMI is that it is a well-understood technology, and numerous examples of scripts can be found on the Internet. With improved support for WMI in Windows PowerShell 2.0, there is very little that cannot be accomplished via PowerShell that can be done from inside VBScript. Before you look at some of the issues in working with WMI from Windows PowerShell, let's review some basic WMI concepts.

WMI is sometimes referred to as a *hierarchical namespace*—so named because the layers build on one another like a Lightweight Directory Access Protocol (LDAP) directory used in Active Directory or the file system structure on your hard disk drive. Although it is true that WMI is a hierarchical namespace, the term doesn't really convey its richness. The WMI model contains three sections: resources, infrastructure, and consumers. The use of these components is found in the following list:

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

Working with Objects and Namespaces

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

The WMI namespace is organized in a similar fashion. The namespaces are the file cabinets. The providers are drawers in the file cabinet. The folders in the drawers of the file cabinet are the WMI classes. These namespaces are shown in Figure 7-1.

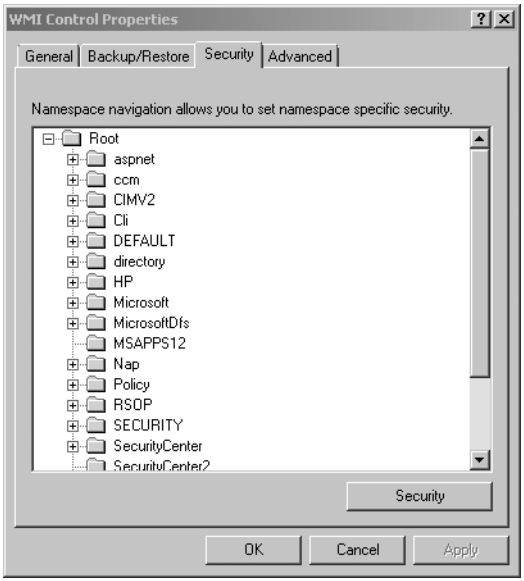


FIGURE 7-1 WMI namespaces on Windows Vista

Namespaces contain objects, and these objects contain properties that you can manipulate. Let's use a WMI command to illustrate how the WMI namespace is organized. The `Get-WmiObject` cmdlet is used to make the connection into the WMI. The class argument is used to specify the *__Namespace* class, and the namespace argument is used to specify the level in the WMI namespace hierarchy. The `Get-WmiObject` line of code is shown here.

```
Get-WmiObject –class __Namespace -namespace root |
Select-Object –property name
```

When the previous code is run, the following result appears on a Windows Vista computer.

```
name
----
subscription
DEFAULT
MicrosoftDfs
CIMV2
Cli
nap
SECURITY
SecurityCenter2
RSOP
WMI
```

```
directory
Policy
ServiceModel
SecurityCenter
Microsoft
aspnet
```

You can use the RecursiveWMINameSpaceListing.ps1 script to get an idea of the number and variety of WMI namespaces that exist on your computer, which is a great way to explore and learn about WMI. The entire contents of the RecursiveWMINameSpaceListing.ps1 script is shown here.

RecursiveWMINameSpaceListing.ps1

```
Function Get-WmiNameSpace($namespace, $computer)
{
 Get-WmiObject -class __NameSpace -computer $computer `
 -namespace $namespace -ErrorAction "SilentlyContinue" |
 Foreach-Object `
 -Process `
   {
     $subns = Join-Path -Path $_.__namespace -ChildPath $_.name
     $subns
     $script:i ++
     Get-WmiNameSpace -namespace $subNS -computer $computer
   }
} #end Get-WmiNameSpace

# *** Entry Point ***

$script:i = 0
$namespace = "root"
$computer = "LocalHost"
"Obtaining WMI Namespaces from $computer ..."
Get-WmiNameSpace -namespace $namespace -computer $computer
"There are $script:i namespaces on $computer"
```

The output from the RecursiveWMINameSpaceListing.ps1 script is shown here from the same Windows Vista computer that produced the earlier namespace listing. You can see that there is a rather intricate hierarchy of namespaces that exists on a modern operating system.

```
Obtaining WMI Namespaces from LocalHost ...
ROOT\subscription
ROOT\subscription\ms_409
ROOT\DEFAULT
ROOT\DEFAULT\ms_409
ROOT\MicrosoftDfs
ROOT\MicrosoftDfs\ms_409
```

```
ROOT\CIMV2
ROOT\CIMV2\Security
ROOT\CIMV2\Security\MicrosoftTpm
ROOT\CIMV2\ms_409
ROOT\CIMV2\TerminalServices
ROOT\CIMV2\TerminalServices\ms_409
ROOT\CIMV2\Applications
ROOT\CIMV2\Applications\Games
ROOT\Cli
ROOT\Cli\MS_409
ROOT\nap
ROOT\SECURITY
ROOT\SecurityCenter2
ROOT\RSOP
ROOT\RSOP\User
ROOT\RSOP\User\S_1_5_21_540299044_341859138_929407116_1133
ROOT\RSOP\User\S_1_5_21_540299044_341859138_929407116_1129
ROOT\RSOP\User\S_1_5_21_540299044_341859138_929407116_1118
ROOT\RSOP\User\S_1_5_21_918056312_2952985149_2686913973_500
ROOT\RSOP\User\S_1_5_21_135816822_1724403450_2350888535_500
ROOT\RSOP\User\ms_409
ROOT\RSOP\User\S_1_5_21_540299044_341859138_929407116_500
ROOT\RSOP\Computer
ROOT\RSOP\Computer\ms_409
ROOT\WMI
ROOT\WMI\ms_409
ROOT\directory
ROOT\directory\LDAP
ROOT\directory\LDAP\ms_409
ROOT\Policy
ROOT\Policy\ms_409
ROOT\ServiceModel
ROOT\SecurityCenter
ROOT\Microsoft
ROOT\Microsoft\HomeNet
ROOT\aspnet
There are 42 namespaces on LocalHost
```

So, what does all of this mean? It means that, on a Windows Vista machine, there are dozens of different namespaces from which you can pull information about your computer. Understanding that the different namespaces exist is the first step to begin navigating in WMI to find the information you need. Often, students and people who are new to Windows PowerShell work on a WMI script to make the script perform a certain action, which is a great way to learn scripting. However, what they often do not know is which namespace they need to connect to so that they can accomplish their task. When I tell them which namespace to work with, they sometimes reply, "It is fine for you, but how do I know that the such and such

namespace even exists?" By using the RecursiveWMINameSpaceListing.ps1 script, you can easily generate a list of namespaces installed on a particular machine and, armed with that information, search MSDN to find out information about those namespaces. Or, if you like to explore, you can move on to the next topic: WMI providers.

Listing WMI Providers

Understanding the namespace assists the network administrator with judiciously applying WMI scripting to his or her network duties. However, as mentioned earlier, to access information via WMI, you must have access to a WMI provider. Once the provider is implemented, you can gain access to the information that is made available. If you want to know which classes are supported by the RouteProvider, you can click the Filter button and select RouteProvider as shown in Figure 7-2.

ON THE COMPANION MEDIA **Two Microsoft Office Excel spreadsheets with all of the providers and their associated classes from Windows XP and Windows Server 2003 are in the Job Aids folder on the companion media.**

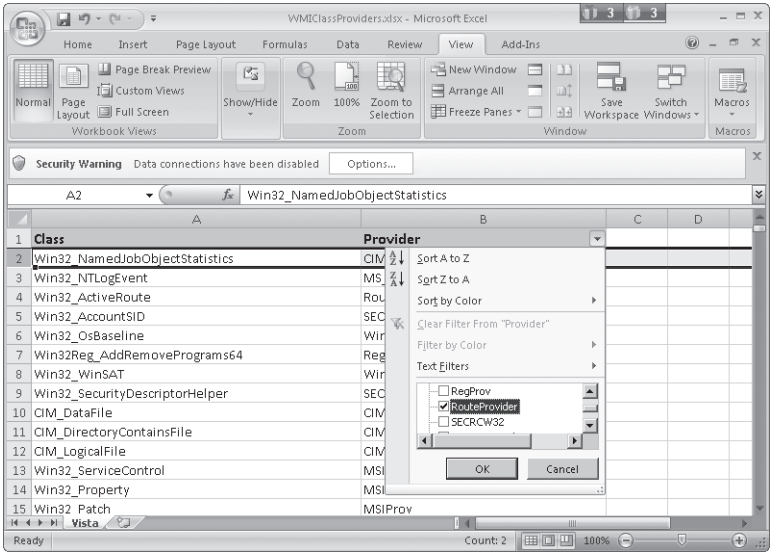


FIGURE 7-2 The WMIProviders spreadsheet lists classes supported by provider name.

Providers in WMI are all based on a template class or on a system class named *__provider*. With this information, you can look for instances of the *__provider* class and obtain a list of all providers that reside in your WMI namespace, which is exactly what the GetWMIProviders.ps1 script accomplishes.

The GetWMIProviders.ps1 script begins by assigning the string "root\cimv2" to the *$wmiNS* variable. This value is used with the `Get-WmiObject` cmdlet to specify where the WMI query takes place. It should be noted that the WMI *root\cimv2* namespace is the default WMI namespace on every Windows operating system since Microsoft Windows 2000.

The `Get-WmiObject` cmdlet is used to query WMI. The class provider is used to limit the WMI query to the *__provider* class. The namespace argument tells the `Get-WmiObject` cmdlet to look only in the *root\cimv2* WMI namespace. The array of objects returned from the *Get-WmiObject* cmdlet is pipelined into the `Sort-Object` cmdlet, where the listing of objects is alphabetized based on the *name* property. Once this process is complete, the reorganized objects are then passed to the `Format-List` cmdlet where the name of each provider is printed. The complete Get-WmiProviders.ps1 script is shown here.

Get-WmiProviders.ps1

```
Function Get-WmiProviders(
                            $namespace="root\cimv2",
                            $computer="localhost"
                           )
{
 Get-WmiObject -class __Provider -namespace $namespace `
 -computername $computer |
 Sort-Object -property Name |
 Select-Object -property Name
} #end Get-WmiProviders

Get-WmiProviders
```

Working with WMI Classes

In addition to working with namespaces, the inquisitive network administrator will also want to explore the concept of classes. In WMI parlance, there are core classes, common classes, and dynamic classes. *Core classes* represent managed objects that apply to all areas of management. These classes provide a basic vocabulary for analyzing and describing managed systems. Two examples of core classes are parameters and the *System.Security* class. *Common classes* are extensions to the core classes and represent managed objects that apply to specific management areas. However, common classes are independent of a particular implementation or technology. *CIM_UnitaryComputerSystem* is an example of a common class. Core and common classes are not used as often by network administrators because they serve as templates from which other classes are derived.

Therefore, many of the classes stored in *root\cimv2* are abstract classes and are used as templates. However, a few classes in *root\cimv2* are dynamic classes that are used to retrieve actual information. What is important to remember about *dynamic classes* is that instances

of a dynamic class are generated by a provider and are therefore more likely to retrieve "live" data from the system.

To produce a simple listing of WMI classes, you can use the `Get-WMIObject` cmdlet and specify the list argument as shown here.

```
Get-WmiObject –list
```

A partial output from the previous command is shown here.

```
Win32_TSGeneralSetting                  Win32_TSPermissionsSetting
Win32_TSClientSetting                   Win32_TSEnvironmentSetting
Win32_TSNetworkAdapterListSetting       Win32_TSLogonSetting
Win32_TSSessionSetting                  Win32_DisplayConfiguration
Win32_COMSetting                        Win32_ClassicCOMClassSetting
Win32_DCOMApplicationSetting            Win32_MSIResource
Win32_ServiceControl                    Win32_Property
```

NOTES FROM THE FIELD

Working with Services

Clint Huffman, Senior Premier Field Engineer (PFE)
Microsoft Corporation

I travel a great deal, and, unfortunately, the battery life on my laptop isn't spectacular. Therefore, I've spent a fair amount of time discovering which services on my computer are consuming the I/O on my hard drive—most likely the largest consumer of battery power other than my monitor. I identified numerous services that I wouldn't need on a flight such as antivirus software, Windows Search, the Offline Files service, ReadyBoost, and so on. Because I was stopping and starting these services quite often, I decided to script the services.

WMI is a powerful object model that allows scripting languages, such as VBScript and Windows PowerShell, to perform tasks that were once only available to hardened C++ developers. Furthermore, far less code is needed to perform these tasks when scripting them makes automation relatively easy.

So, to begin this script, I need to select the correct services. WMI uses a SQL-like syntax named WMI Query Language (WQL); it is not named SQL syntax because WQL has some odd quirks that are specific to WMI. Now, I want my WQL query to return the Windows services that I identified earlier as users of frequent disk I/O such as the Offline Files service, the ReadyBoost service, my antivirus services that begin with "Microsoft ForeFront" (Microsoft Forefront Client Security Antimalware Service and Microsoft Forefront Client Security State Assessment Service), and lastly, my personal file indexer, Windows Search.

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
Get-WmiObject -Query $WQL
In my case, this script returns the following services as:
Offline Files
ReadyBoost
Microsoft Forefront Client Security Antimalware Service
Microsoft Forefront Client Security State Assessment Service
Windows Search
```

The *Caption* property is the text you see when you bring up Control Panel, Services, and the *name* property is the short name of the service that you might be more familiar with when using the command-line tool "Net Start" and "Net Stop." Finally, the *State* property tells me whether the service is running.

The WHERE clause allows me to limit the information that is returned. For example, if I don't use the WHERE clause, I receive all of the services as objects. This is nice if you want to know what services are on a computer, but it's not helpful when you simply want to shut down a few of them. For more information about WQL, go to "Querying with WQL" at *http://msdn.microsoft.com/en-us/library/aa392902.aspx*.

Because the *Query* parameter always returns a *collection* object, I need to enumerate the *Query* parameter to work with each item individually. This process is similar to receiving a package in the mail in a large cardboard box: before I can use what's inside, I need to open the package first. This is the point in the process in which the *Foreach* flow control statement is used. The *Foreach* statement allows me to work with one item at a time (for example, a service), which is similar to taking one item out of the cardboard box at a time. In this case, I have the `Get-WmiObject` cmdlet's return values go into a variable named *$CollectionOfServices* (my cardboard box). Next, I use the *Foreach* statement to work with each service, whereby the *$Service* variable becomes each service object in turn. The following code is the same as the previous code but with the addition of a *Foreach* loop.

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL
Foreach ($Service in $CollectionOfServices)
{
  $Service.Caption
}
```

Now that I can select specific services that I want to shut down, let's actually shut them down. I can do this by using the *StopService()* method as follows:

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL
Foreach ($Service in $CollectionOfServices)
{
  $Service.Caption
  $Service.StopService()
}
```

If my services don't actually stop, it is most likely because I don't have administrator rights to my customer or, if I am on Windows Vista, I need to run the script in an elevated Windows PowerShell command prompt. To make an elevated Windows PowerShell command prompt, right-click on the PowerShell icon, select Run As Administrator, and then try the script again.

Great! My unnecessary services are stopped. However, sometimes the services can be a bit tenacious and start up again the first chance they get. How do I hold them down? By setting them to disabled. How do I do that? By using the *ChangeStartMode()* method with the argument/parameter of "Disabled" as follows:

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL
Foreach ($Service in $CollectionOfServices)
{
  $Service.Caption
  $Service.StopService()
  $Service.ChangeStartMode("Disabled")
}
```

Now we're talking! Those pesky services are down for the count.

I've had my fun, my flight is over, and now I need to connect to my corporate network. Corporate policy does not allow me to connect unless my antivirus service is running. No problem. Two slight modifications to the script and the services are running again as follows:

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL
Foreach ($Service in $CollectionOfServices)
{
  $Service.Caption
  $Service.StartService()
```

```
  $Service.ChangeStartMode("Automatic")
}
```

I replaced the *StopService()* method with *StartService()* and replaced the argument of the *ChangeStartMode()* method to "Automatic."

You might be thinking that this procedure is all well and good for your laptop battery, but what about doing massive restarts of services? Well, a great modification that you can make to the script is to run it against remote servers. For example, let's assume that you need to restart the services in a farm of 10 Web servers. You can simply modify the script slightly by adding the *–ComputerName* argument.

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL -ComputerName
demoserver
Foreach ($Service in $CollectionOfServices)
{
  $Service.Caption
  $Service.StartService()
  $Service.ChangeStartMode("Automatic")
}
```

These scripts have served me well, and I hope they help you too.

Changing Settings

For all of the benefits of using WMI, there are still many frustrating limitations. While WMI is good at retrieving information, it is not always very good at changing that information. The following example illustrates this point. The *Win32_Desktop* WMI class provides information about desktop settings as shown here.

```
PS C:\> Get-WmiObject Win32_Desktop

__GENUS                : 2
__CLASS                : Win32_Desktop
__SUPERCLASS           : CIM_Setting
__DYNASTY              : CIM_Setting
__RELPATH              : Win32_Desktop.Name="NT AUTHORITY\\SYSTEM"
__PROPERTY_COUNT       : 21
__DERIVATION           : {CIM_Setting}
__SERVER               : MRED1
__NAMESPACE            : root\cimv2
```

```
__PATH                : \\MRED1\root\cimv2:Win32_Desktop.Name="NT AUTHORITY\\SY
                        STEM"
BorderWidth           : 1
Caption               :
CoolSwitch            :
CursorBlinkRate       : 500
Description           :
DragFullWindows       : True
GridGranularity       :
IconSpacing           :
IconTitleFaceName     : Segoe UI
IconTitleSize         : 9
IconTitleWrap         : True
Name                  : NT AUTHORITY\SYSTEM
Pattern               : (None)
ScreenSaverActive     : True
ScreenSaverExecutable : C:\Windows\system32\logon.scr
ScreenSaverSecure     : True
ScreenSaverTimeout    : 600
SettingID             :
Wallpaper             :
WallpaperStretched    : False
WallpaperTiled        :
```

As you can see from the properties and values that are returned from the `Get-WmiObject` cmdlet, much of the information is valuable. Items such as screen saver time-out values and secure screen saver are routine concerns to many network administrators. While it is true that these values can, and in most cases should, be set via Group Policy, there are times when network administrators want the ability to change these values via script. If you use the `Get-Member` cmdlet to examine the properties of the *Win32_Desktop* WMI class, you are greeted with the following information.

```
PS C:\> Get-WmiObject Win32_Desktop | Get-Member
   TypeName: System.Management.ManagementObject#root\cimv2\Win32_Desktop

Name                MemberType   Definition
----                ----------   ----------
BorderWidth         Property     System.UInt32 BorderWidth {get;set;}
Caption             Property     System.String Caption {get;set;}
CoolSwitch          Property     System.Boolean CoolSwitch {get;set;}
CursorBlinkRate     Property     System.UInt32 CursorBlinkRate {get;set;}
Description         Property     System.String Description {get;set;}
DragFullWindows     Property     System.Boolean DragFullWindows {get;set;}
GridGranularity     Property     System.UInt32 GridGranularity {get;set;}
IconSpacing         Property     System.UInt32 IconSpacing {get;set;}
IconTitleFaceName   Property     System.String IconTitleFaceName {get;set;}
IconTitleSize       Property     System.UInt32 IconTitleSize {get;set;}
```

```
IconTitleWrap         Property     System.Boolean IconTitleWrap {get;set;}
Name                  Property     System.String Name {get;set;}
Pattern               Property     System.String Pattern {get;set;}
ScreenSaverActive     Property     System.Boolean ScreenSaverActive {get;set;}
ScreenSaverExecutable Property     System.String ScreenSaverExecutable {get;...
ScreenSaverSecure     Property     System.Boolean ScreenSaverSecure {get;set;}
ScreenSaverTimeout    Property     System.UInt32 ScreenSaverTimeout {get;set;}
SettingID             Property     System.String SettingID {get;set;}
Wallpaper             Property     System.String Wallpaper {get;set;}
WallpaperStretched    Property     System.Boolean WallpaperStretched {get;set;}
WallpaperTiled        Property     System.Boolean WallpaperTiled {get;set;}
__CLASS               Property     System.String __CLASS {get;set;}
__DERIVATION          Property     System.String[] __DERIVATION {get;set;}
__DYNASTY             Property     System.String __DYNASTY {get;set;}
__GENUS               Property     System.Int32 __GENUS {get;set;}
__NAMESPACE           Property     System.String __NAMESPACE {get;set;}
__PATH                Property     System.String __PATH {get;set;}
__PROPERTY_COUNT      Property     System.Int32 __PROPERTY_COUNT {get;set;}
__RELPATH             Property     System.String __RELPATH {get;set;}
__SERVER              Property     System.String __SERVER {get;set;}
__SUPERCLASS          Property     System.String __SUPERCLASS {get;set;}
ConvertFromDateTime   ScriptMethod System.Object ConvertFromDateTime();
ConvertToDateTime     ScriptMethod System.Object ConvertToDateTime();
```

When you use the *–filter* parameter to obtain a specific instance of the *Win32_Desktop* WMI class and store it in a variable, you can then directly access the properties of the class. In this example, you need to escape the backslash that is used as a separator between NT Authority and System as shown here.

```
PS C:\> $desktop = Get-WmiObject Win32_Desktop -Filter `
>> "name = 'NT AUTHORITY\\SYSTEM'"
```

Once you have access to a specific instance of the WMI class, you can then assign a new value for the *ScreenSaverTimeout* parameter. As shown here, the value is updated immediately.

```
PS C:\> $Desktop.ScreenSaverTimeout = 300
PS C:\> $Desktop.ScreenSaverTimeout
300
```

However, if you resubmit the WMI query, you see that the *ScreenSaverTimeout* property is not updated. The get:set that is reported by the `Get-Member` cmdlet is related to the copy of the object that is returned by the WMI query and not to the actual instance of the object represented by the WMI class as shown here.

```
PS C:\> $desktop = Get-WmiObject Win32_Desktop -Filter `
>> "name = 'NT AUTHORITY\\SYSTEM'"
>>
PS C:\> $Desktop.ScreenSaverTimeout
600
```

Modifying Values Through the Registry

The GetSetScreenSaverTimeOut.ps1 script uses a single parameter named *debug*. This parameter is a *switched parameter*, which means that it only performs a function when it is present. The script prints detailed information when you run the script with the *debug* switch, such as letting you know which function is currently being called as shown in Figure 7-3.

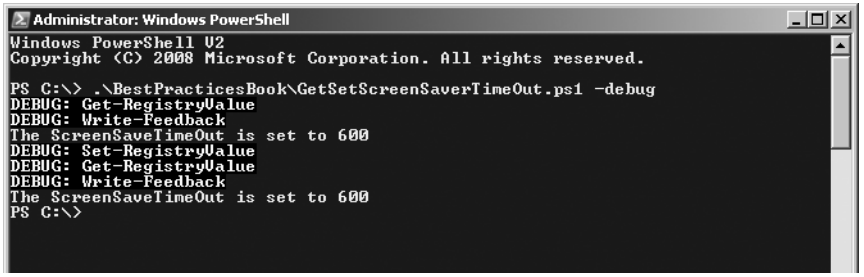


FIGURE 7-3 Detailed debug information is easily obtained when the script implements a *debug* parameter.

To create a command-line parameter, you can use the *Param* statement as shown here.

```
Param([switch]$debug)
```

Following the *Param* statement, which needs to be the first noncommented line of code in the script, the *Get-RegistryValue* function is created. In this code, the *$in* variable is passed by reference, which means that the function assigns a new value to the variable. This value is used outside of the function that assigns the value to it. To pass the variable by reference, you need to convert the *$in* variable to a reference type; you can use the [ref] type to perform this conversion. Therefore, you need to create the *$in* variable prior to calling the function because you cannot cast the variable to a reference type if it does not exist as shown here.

```
Function Get-RegistryValue([ref]$in)
```

Now you come to the first `Write-Debug` cmdlet. To write the debug information to the console prompt, the script uses the `Write-Debug` cmdlet. The `Write-Debug` cmdlet automatically formats the text with yellow and black colors (this is configurable, however), and it only writes text to the console if you tell it to do so. By default, `Write-Debug` does not print anything to the console, which means that you do not need to remove the `Write-Debug` statements prior to deploying the script. The *$DebugPreference* automatic variable is used to control the behavior of the `Write-Debug` cmdlet. By default, *$DebugPreference* is set to *SilentlyContinue* so that when it encounters a `Write-Debug` cmdlet, Windows PowerShell either skips over the cmdlet or silently continues to the next line. You can configure the *$DebugPreference* variable with one of four values defined in the *System.Management.Automation.ActionPreference* enumeration class. To see the possible enumeration values, you can either look for them on MSDN or use the *GetNames* static method from the *System.Enum* .NET Framework class as shown here.

```
PS C:\> [enum]::GetNames("System.Management.Automation.ActionPreference")
SilentlyContinue
Stop
Continue
Inquire
```

The `Write-Debug` cmdlet is used to print the value of the *name* property from the *System.Management.Automation.ScriptInfo* object. The *System.Management.Automation.ScriptInfo* object is obtained by querying the *MyCommand* property of the *System.Management.Automation.InvocationInfo* class. A *System.Management.Automation.InvocationInfo* object is returned when you query the *$MyInvocation* automatic variable. The properties of *System.Management.Automation.InvocationInfo* are shown in Table 7-2.

TABLE 7-2 Properties of the *System.Management.Automation.InvocationInfo* Class

PROPERTY	DEFINITION
BoundParameters	System.Collections.Generic.Dictionary`2[[System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.Object, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]] BoundParameters {get;}
CommandOrigin	System.Management.Automation.CommandOrigin CommandOrigin {get;}
ExpectingInput	System.Boolean ExpectingInput {get;}
InvocationName	System.String InvocationName {get;}
Line	System.String Line {get;}
MyCommand	System.Management.Automation.CommandInfo MyCommand {get;}
OffsetInLine	System.Int32 OffsetInLine {get;}
PipelineLength	System.Int32 PipelineLength {get;}
PipelinePosition	System.Int32 PipelinePosition {get;}
PositionMessage	System.String PositionMessage {get;}
ScriptLineNumber	System.Int32 ScriptLineNumber {get;}
ScriptName	System.String ScriptName {get;}
UnboundArguments	System.Collections.Generic.List`1[[System.Object, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]] UnboundArguments {get;}

The `Write-Debug` commands can be modified to include any of the properties you deem helpful to aid in troubleshooting. These properties become even more helpful when you are working with the *System.Management.Automation.ScriptInfo* object, whose properties are shown in Table 7-3.

TABLE 7-3 Properties of the *System.Management.Automation.ScriptInfo* Object

PROPERTY	DEFINITION
CommandType	System.Management.Automation.CommandTypes CommandType {get;}
Definition	System.String Definition {get;}
Module	System.Management.Automation.PSModuleInfo Module {get;}
ModuleName	System.String ModuleName {get;}
Name	System.String Name {get;}
Parameters	System.Collections.Generic.Dictionary`2[[System.String, mscorlib,Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089], [System.Management.Automation.ParameterMetadata, System.Management.Automation, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]] Parameters {get;}
ParameterSets	System.Collections.ObjectModel.ReadOnlyCollection`1 [[System.Management.Automation.CommandParameterSetInfo, System.Management.Automation, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]] ParameterSets {get;}
ScriptBlock	System.Management.Automation.ScriptBlock ScriptBlock {get;}
Visibility	System.Management.Automation.SessionStateEntryVisibility Visibility {get;set;}

The `Write-Debug` command is shown here.

```
{
Write-Debug $MyInvocation.MyCommand.name
```

To use the *$in* reference type variable, you must assign the data to the *Value* property of the variable. The `Get-ItemProperty` cmdlet creates a custom Windows PowerShell object. As you can see here, a number of properties are contained in the custom object.

```
PS C:\> $swValue = Get-ItemProperty -Path HKCU:\Scripting\Stopwatch
PS C:\> $swValue
PSPath          : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Scripting\
Stopwatch
PSParentPath    : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Scripting
PSChildName     : Stopwatch
PSDrive         : HKCU
PSProvider      : Microsoft.PowerShell.Core\Registry
PreviousCommand : 00:00:00.3153793
```

You do not have to use the intermediate variable to obtain the previous *Value* property. You can use parentheses and query the property directly. Although this may be a bit confusing, it is certainly a valid syntax as shown here.

```
PS C:\> (Get-ItemProperty -Path HKCU:\Scripting\Stopwatch).PreviousCommand
00:00:00.3153793
```

Once you have the custom object, you can query the *name* property and assign it to the *Value* property of the *$in* reference type variable as shown here.

```
 $in.value = (Get-ItemProperty -path $path -name $name).$name
} #end Get-RegistryValue
```

Next, the *Set-RegistryValue* function is created. This function accepts an input variable named *$value*. As in the previous function, you first use the `Write-Debug` cmdlet to print the name of the function. Then, the `Set-ItemProperty` cmdlet is used to assign the value to the registry. This new value was passed when the function was called and is contained in the *$value* variable as shown here.

```
Function Set-RegistryValue($value)
{
 Write-Debug $MyInvocation.MyCommand.name
 Set-ItemProperty -Path $path -name $name -value $value
} #end Get-RegistryValue
```

Once the registry is updated via the *Set-RegistryValue* function, it is time to provide feedback to the user via the *Write-Feedback* function. The `Write-Debug` cmdlet is used to print debug information stating that the script is in the *Set-RegistryValue* function. This information is displayed only when the script is run with the *–debug* switch. The next line of the script is always used to display feedback. One interesting item is the *subexpression*, which is used to force the evaluation of the *Value* property to return the reference type object. This may seem a bit confusing until you understand that there are two types of string characters in Windows PowerShell. The first is a literal string, which is represented with single quotation marks. In a literal string, what you see is what you get and a variable is not expanded inside single quotation marks as shown here.

```
PS C:\> $a = "this is a string"
PS C:\> 'This is what is in $a'
This is what is in $a
```

When you use an expanding string, which is represented by double quotation marks, the value of the variable is expanded and printed as shown here.

```
PS C:\> $a = "this is a string"
PS C:\> "This is what is in $a"
This is what is in this is a string
```

While frustrating at first, the expanding string behavior can be used to your advantage to avoid concatenation of strings and variables. Once you are aware of the two strings, you can use the backtick character to suppress variable expansion when desired and proceed as follows:

```
PS C:\> $a = "this is a string"
PS C:\> "This is what is in `$a: $a"
This is what is in $a: this is a string
```

If you do not have expanding strings, you need to concatenate the output as shown here.

```
PS C:\> $a = "this is a string"
PS C:\> 'This is what is in $a: ' + $a
This is what is in $a: this is a string
```

So what does the expanding string behavior have to do with your code? When an object is expanded in an expanding string, it tells you the name of the object. If you want to see the value, you need to create a subexpression by placing the object in smooth parentheses and placing a dollar sign in front of it. This action forces evaluation of the object and returns the default property to the command line as shown here.

```
Function Write-Feedback($in)
{
 Write-Debug $MyInvocation.MyCommand.name
"The $name is set to $($in)"
} #end Write-Feedback
```

Next, the script checks to determine whether the *$debug* variable exists. If the *$debug* variable exists, it means the script was launched with the *–debug* switch. If this is the case, the script changes the value of the *$debugPreference* automatic variable from *SilentlyContinue* to *continue*. This action causes the *debug* statements created by the `Write-Debug` cmdlet to be emitted to the command line as shown here.

```
if($debug) { $DebugPreference = "continue" }
```

Now it is time to initialize several variables. The first variable is the path to the desktop settings, and the next variable is the name of the registry key to change. These two variables are shown here.

```
$path = 'HKCU:\Control Panel\Desktop'
$name = 'ScreenSaveTimeOut'
```

The last two variables that need to be initialized are the *$in* variable and the value assigned to the screen saver time-out as shown here.

```
$in = $null
$value = 600
```

The remainder of the script calls the functions in correct order. The first function called is the *Get-RegistryValue* function, which obtains the current value of the screen saver time-out. The *Write_Feedback* function is called to print the value of the screen saver time-out. Next, the *Set-RegistryValue* function is called, which updates the screen saver time-out. The *Get-RegistryValue* function is then called to obtain the registry value, and it is displayed with the *Write-Feedback* function as shown here.

```
Get-RegistryValue([ref]$in)
Write-Feedback($in)
Set-RegistryValue($value)
Get-RegistryValue([ref]$in)
Write-Feedback($in)
```

The completed GetSetScreenSaverTimeOut.ps1 is shown here.

GetSetScreenSaverTimeOut.ps1

```
Param([switch]$debug)
Function Get-RegistryValue([ref]$in)
{
 Write-Debug $MyInvocation.MyCommand.name
 $in.value = (Get-ItemProperty -path $path -name $name).$name
} #end Get-RegistryValue

Function Set-RegistryValue($value)
{
 Write-Debug $MyInvocation.MyCommand.name
 Set-ItemProperty -Path $path -name $name -value $value
} #end Get-RegistryValue

Function Write-Feedback($in)
{
 Write-Debug $MyInvocation.MyCommand.name
 "The $name is set to $($in)"
} #end Write-Feedback

# *** Entry Point ***
if($debug) { $DebugPreference = "continue" }
$path = 'HKCU:\Control Panel\Desktop'
$name = 'ScreenSaveTimeOut'
$in = $null
$value = 600

Get-RegistryValue([ref]$in)
Write-Feedback($in)
Set-RegistryValue($value)
Get-RegistryValue([ref]$in)
Write-Feedback($in)
```

Lack of .NET Framework Support

The ability to work with the .NET Framework from within Windows PowerShell is very exciting. Because Windows PowerShell itself is a .NET Framework application, access to the .NET Framework is very direct and natural. At times, the question is not what can be done with .NET Framework classes, but rather what cannot be done. The constructors for some of the .NET Framework classes can be both confusing and complicated. A *constructor* is used to create an instance of a class; in many cases, you must first create an instance of a class prior to using the classes. However, sometimes you do not need a constructor at all, and these methods are called static. There are both static methods and static properties.

Use of Static Methods and Properties

Static methods and properties are members that are always available. To use a static method, you place the class name in square brackets and separate the method name by two colons. An example is the *tan* method from the *System.Math* class. The *tan* method is used to find the tangent of a number. As shown here, you can use the *tan* static method from the *System.Math* class to find the tangent of a 45-degree angle.

```
PS C:\> [system.math]::tan(45)
1.61977519054386
```

When referring to *System.Math*, the word *system* is used to represent the namespace in which the "*math* class" is found. In most cases, you can drop the word *system* if you want to and the process will work exactly the same. When working at the command line, you may want to save some typing and drop the word *system*, but I consider it to be a best practice to always include the word *system* in a script. If you drop the word *system*, the command looks like the following code.

```
PS C:\> [math]::tan(45)
1.61977519054386
```

You can use the `Get-Member` cmdlet with the *static* switched parameter to obtain the members of the *System.Math* .NET Framework class. To do this, the command looks like the following example.

```
[math] | Get-Member –static
```

The static members of the *System.Math* class are shown in Table 7-4. These static methods are very important because you can perform most of the functionality from the class by using them. For example, there is no *tan* function built into Windows PowerShell. If you want the tangent of an angle, you must use either the static methods from *System.Math* or write your own tangent function. This occurs by design. To perform these mathematical computations, you need to use the .NET Framework. Rather than being a liability, the .NET Framework is a tremendous asset because it is a mature technology and is well documented.

TABLE 7-4 Members of the *System.Math* Class

NAME	MEMBERTYPE	DEFINITION
Abs	Method	static System.SByte Abs(SByte value) static System.Int16 Abs(Int16 value) static System.Int32 Abs(Int32 value) static System.Int64 Abs(Int64 value) static System.Single Abs(Single value) static System.Double Abs(Double value) static System.Decimal Abs(Decimal value)
Acos	Method	static System.Double Acos(Double d)
Asin	Method	static System.Double Asin(Double d)
Atan	Method	static System.Double Atan(Double d)
Atan2	Method	static System.Double Atan2(Double y Double x)
BigMul	Method	static System.Int64 BigMul(Int32 a Int32 b)
Ceiling	Method	static System.Decimal Ceiling(Decimal d) static System.Double Ceiling(Double a)
Cos	Method	static System.Double Cos(Double d)
Cosh	Method	static System.Double Cosh(Double value)
DivRem	Method	static System.Int32 DivRem(Int32 a Int32 b Int32& result) static System.Int64 DivRem(Int64 a Int64 b Int64& result)
Equals	Method	static System.Boolean Equals(Object objA Object objB)
Exp	Method	static System.Double Exp(Double d)
Floor	Method	static System.Decimal Floor(Decimal d) static System.Double Floor(Double d)
IEEERemainder	Method	static System.Double IEEERemainder(Double x Double y)
Log	Method	static System.Double Log(Double d) static System.Double Log(Double a Double newBase)
Log10	Method	static System.Double Log10(Double d)
Max	Method	static System.SByte Max(SByte val1 SByte val2) static System.Byte Max(Byte val1 Byte val2) static System.Int16 Max(Int16 val1 Int16 val2) static System.UInt16 Max(UInt16 val1 UInt16 val2) static System.Int32 Max(Int32 val1 Int32 val2) static System.UInt32 Max(UInt32 val1 UInt32 val2) static System.Int64 Max(Int64 val1 Int64 val2) static System.UInt64 Max(UInt64 val1 UInt64 val2) static System.Single Max(Single val1 Single val2) static System.Double Max(Double val1 Double val2) static System.Decimal Max(Decimal val1 Decimal val2)

NAME	MEMBERTYPE	DEFINITION
Min	Method	static System.SByte Min(SByte val1 SByte val2) static System.Byte Min(Byte val1 Byte val2) static System.Int16 Min(Int16 val1 Int16 val2) static System.UInt16 Min(UInt16 val1 UInt16 val2) static System.Int32 Min(Int32 val1 Int32 val2) static System.UInt32 Min(UInt32 val1 UInt32 val2) static System.Int64 Min(Int64 val1 Int64 val2) static System.UInt64 Min(UInt64 val1 UInt64 val2) static System.Single Min(Single val1 Single val2) static System.Double Min(Double val1 Double val2) static System.Decimal Min(Decimal val1 Decimal val2)
Pow	Method	static System.Double Pow(Double x Double y)
ReferenceEquals	Method	static System.Boolean ReferenceEquals(Object objA Object objB)
Round	Method	static System.Double Round(Double a) static System.Double Round(Double value Int32 digits) static System.Double Round(Double value MidpointRounding mode) static System.Double Round(Double value Int32 digits MidpointRounding mode) static System.Decimal Round(Decimal d) static System.Decimal Round(Decimal d Int32 decimals) static System.Decimal Round(Decimal d MidpointRounding mode) static System.Decimal Round(Decimal d Int32 decimals MidpointRounding mode)
Sign	Method	static System.Int32 Sign(SByte value) static System.Int32 Sign(Int16 value) static System.Int32 Sign(Int32 value) static System.Int32 Sign(Int64 value) static System.Int32 Sign(Single value) static System.Int32 Sign(Double value) static System.Int32 Sign(Decimal value)
Sin	Method	static System.Double Sin(Double a)
Sinh	Method	static System.Double Sinh(Double value)
Sqrt	Method	static System.Double Sqrt(Double d)
Tan	Method	static System.Double Tan(Double a)
Tanh	Method	static System.Double Tanh(Double value)
Truncate	Method	static System.Decimal Truncate(Decimal d) static System.Double Truncate(Double d)
E	Property	static System.Double E {get;}
PI	Property	static System.Double PI {get;}

Version Dependencies

One of the more interesting facets of the .NET Framework is that there always seems to be a new version available, and, of course, between versions there are service packs. While the .NET Framework is included in the operating system, updates to the .NET Framework are unfortunately not included in service packs. It therefore becomes the responsibility of the network administrators to package and deploy updates to the framework. Until the introduction of Windows PowerShell, network administrators were not keen to provide updates simply because they did not have a vested interest in the deployment of the .NET Framework. This behavior was not due to a lack of interest; in many cases, it was due to a lack of understanding of the .NET Framework. If developers did not request updates to the .NET Framework, then it did not get updated.

Lack of COM Support

Many very useful capabilities are packaged as Component Object Model (COM) components. Finding these COM objects is sometimes a matter of luck. Of course, you can always read the MSDN documentation; unfortunately, the articles do not always list the program ID that is required to create the COM object, and this is even true in articles that refer to the scripting interfaces. An example can be found in the Windows Media Player scripting object model. You can work your way through the entire Software Development Kit (SDK) documentation without discovering that the program ID is *wmplayer.ocx* and not *player*, which is used for illustrative purposes. The most natural way to work with a COM object in Windows PowerShell is to use the `New-Object` cmdlet, specify the *–ComObject* parameter, and give the parameter the program ID. If the program ID is not forthcoming, then you have a more difficult proposition. You can search the registry and, by doing a bit of detective work, find the program ID.

An example of a COM object whose program ID is hard to find is the object with the *makecab.makecab* program ID. The *makecab.makecab* object is used to make *cabinet files,* which are highly compressed files often used by programmers to deploy software applications. There is no reason why an enterprise network administrator cannot use .cab files to compress log files prior to transferring them across the application. The only problem is that, while the *makecab.makecab* object is present in Windows XP and Windows Server 2003, it has been removed from the operating system beginning with Windows Vista. When working with newer operating systems, a different approach is required.

To make the script easier to use, you must first create some command-line parameters by using the *Param* statement. The *Param* statement must be the first noncommented line in the script. When the script is run from within the Windows PowerShell console or from within a script editor, the command-line parameters are used to control the way in which the script executes. In this way, the script can be run without needing to edit it each time you want to create a .cab file from a different directory. You only need to supply a new value for the *–filepath* parameter as shown here.

```
CreateCab.ps1 –filepath C:\fso1
```

What is good about command-line parameters is that they use partial parameter completion, which means that you only need to supply enough of the parameter for it to be unique. Therefore, you can use command-line syntax such as the following:

```
CreateCab.ps1 –f c:\fso1 –p c:\fso2\bcab.cab –d
```

The previous syntax searches the c:\fso directory and obtains all of the files. It then creates a cabinet file named bcab.cab in the fso2 folder of the C:\ drive. The syntax also produces debugging information while it is running. Note that the *debug* parameter is a switched parameter because *debug* only affects the script when it is present. This section of the CreateCab.ps1 script is shown here.

```
Param(
      $filepath = "C:\fso",
      $path = "C:\fso\aCab.cab",
      [switch]$debug
     )
```

It is now time to create the *New-Cab* function, which will accept two input parameters. The first is the *–path* parameter, and the second is the *–files* parameter.

```
Function New-Cab($path,$files)
```

You can assign the *makecab.makecab* program ID to a variable named *$makecab*, which makes the script a bit easier to read. This is also a good place to put the first `Write-Debug` statement.

```
{
 $makecab = "makecab.makecab"
 Write-Debug "Creating Cab path is: $path"
```

You now need to create the COM object.

```
 $cab = New-Object -ComObject $makecab
```

A bit of error checking is in order. To do this, you can use the *$?* automatic variable.

```
 if(!$?) { $(Throw "unable to create $makecab object")}
```

If no errors occur during the attempt to create the *makecab.makecab* object, then you can use the object contained in the *$cab* variable and call the *createcab* method.

```
 $cab.CreateCab($path,$false,$false,$false)
```

After you create the .cab file, you need to add files to it by using the *Foreach* statement.

```
 Foreach ($file in $files)
  {
   $file = $file.fullname.tostring()
   $fileName = Split-Path -path $file -leaf
```

After you turn the full file name into a string and remove the directory information by using the `Split-Path` cmdlet, another `Write-Debug` statement is needed to let the user of the script be informed of progress as shown here.

```
   Write-Debug "Adding from $file"
   Write-Debug "File name is $fileName"
```

Next, you need to add a file to the cabinet file.

```
   $cab.AddFile($file,$filename)
  }
 Write-Debug "Closing cab $path"
```

To close the cabinet file, you can use the *closecab* method.

```
 $cab.CloseCab()
} #end New-Cab
```

It is now time to go to the entry point of the script. First, you must determine whether the script is being run in debug mode by looking for the presence of the *$debug* variable. If it is running in debug mode, you must set the value of the *$DebugPreference* variable to *continue*, which allows the `Write-Debug` statements to be printed on the screen. By default, *$DebugPreference* is set to *SilentlyContinue*, which means that no debug statements are displayed and Windows PowerShell skips past the `Write-Debug` command without taking any action as shown here.

```
if($debug) {$DebugPreference = "continue"}
```

Now, you need to obtain a collection of files by using the `Get-ChildItem` cmdlet.

```
$files = Get-ChildItem -path $filePath | Where-Object { !$_.psiscontainer }
```

After you have a collection of files, you can pass the collection to the *New-Cab* function as shown here.

```
New-Cab -path $path -files $files
```

The completed CreateCab.ps1 script is shown here. Note: The CreateCab.ps1 script will not run on Windows Vista and later versions due to lack of support for the *makecab.makecab* COM object. An alternate method of creating .cab files is explored in the "Lack of External Application Support" section later in the chapter.

CreateCab.ps1

```
Param(
      $filepath = "C:\fso",
      $path = "C:\fso\aCab.cab",
      [switch]$debug
     )
Function New-Cab($path,$files)
{
 $makecab = "makecab.makecab"
 Write-Debug "Creating Cab path is: $path"
```

```
 $cab = New-Object -ComObject $makecab
 if(!$?) { $(Throw "unable to create $makecab object")}
 $cab.CreateCab($path,$false,$false,$false)
 Foreach ($file in $files)
  {
   $file = $file.fullname.tostring()
   $fileName = Split-Path -path $file -leaf
   Write-Debug "Adding from $file"
   Write-Debug "File name is $fileName"
   $cab.AddFile($file,$filename)
  }
 Write-Debug "Closing cab $path"
 $cab.CloseCab()
} #end New-Cab

# *** entry point to script ***
if($debug) {$DebugPreference = "continue"}
$files = Get-ChildItem -path $filePath | Where-Object { !$_.psiscontainer }
New-Cab -path $path -files $files
```

You cannot use the *makecab.makecab* object to expand the cabinet file because it does not have an *expand* method. You also cannot use the *makecab.expandcab* object because it does not exist. Because the ability to expand a cabinet file is inherent in the Windows shell, you can use the *shell* object to expand the cabinet file. To access the shell, you can use the *Shell.Application* COM object.

You must first create command-line parameters. This section of the script is very similar to the parameter section of the previous CreateCab.ps1 script. The command-line parameters are shown here.

```
Param(
      $cab = "C:\fso\acab.cab",
      $destination = "C:\fso1",
      [switch]$debug
     )
```

After you create command-line parameters, it is time to create the *ConvertFrom-Cab* function, which will accept two command-line parameters. The first parameter contains the .cab file, and the second parameter contains the destination to expand the files as shown here.

```
Function ConvertFrom-Cab($cab,$destination)
```

You should now create an instance of the *Shell.Application* object. The *Shell.Application* object is a very powerful object with a number of useful methods. The members of the *Shell.Application* object are shown in Table 7-5.

TABLE 7-5 Members of the *Shell.Application* Object

NAME	MEMBERTYPE	DEFINITION
AddToRecent	Method	void AddToRecent (Variant, string)
BrowseForFolder	Method	Folder BrowseForFolder (int, string, int, Variant)
CanStartStopService	Method	Variant CanStartStopService (string)
CascadeWindows	Method	void CascadeWindows ()
ControlPanelItem	Method	void ControlPanelItem (string)
EjectPC	Method	void EjectPC ()
Explore	Method	void Explore (Variant)
ExplorerPolicy	Method	Variant ExplorerPolicy (string)
FileRun	Method	void FileRun ()
FindComputer	Method	void FindComputer ()
FindFiles	Method	void FindFiles ()
FindPrinter	Method	void FindPrinter (string, string, string)
GetSetting	Method	bool GetSetting (int)
GetSystemInformation	Method	Variant GetSystemInformation (string)
Help	Method	void Help ()
IsRestricted	Method	int IsRestricted (string, string)
IsServiceRunning	Method	Variant IsServiceRunning (string)
MinimizeAll	Method	void MinimizeAll ()
NameSpace	Method	Folder NameSpace (Variant)
Open	Method	void Open (Variant)
RefreshMenu	Method	void RefreshMenu ()
ServiceStart	Method	Variant ServiceStart (string, Variant)
ServiceStop	Method	Variant ServiceStop (string, Variant)
SetTime	Method	void SetTime ()
ShellExecute	Method	void ShellExecute (string, Variant, Variant, Variant, Variant)
ShowBrowserBar	Method	Variant ShowBrowserBar (string, Variant)
ShutdownWindows	Method	void ShutdownWindows ()
Suspend	Method	void Suspend ()
TileHorizontally	Method	void TileHorizontally ()
TileVertically	Method	void TileVertically ()

NAME	MEMBERTYPE	DEFINITION
ToggleDesktop	Method	void ToggleDesktop ()
TrayProperties	Method	void TrayProperties ()
UndoMinimizeALL	Method	void UndoMinimizeALL ()
Windows	Method	IDispatch Windows ()
WindowsSecurity	Method	void WindowsSecurity ()
WindowSwitcher	Method	void WindowSwitcher ()
Application	Property	IDispatch Application () {get}
Parent	Property	IDispatch Parent () {get}

Because you want to use the name of the COM object more than once, it is a good practice to assign the program ID of the COM object to a variable. You can then use the string with the `New-Object` cmdlet and also use it when providing feedback to the user. The line of code that assigns the *Shell.Application* program ID to a string is shown here.

```
{
 $comObject = "Shell.Application"
```

It is now time to provide some feedback to the user. You can do this by using the `Write-Debug` cmdlet together with a message stating that you are attempting to create the *Shell.Application* object as shown here.

```
  Write-Debug "Creating $comObject"
```

After you provide debug feedback stating you are going to create the object, you can actually create the object as shown here.

```
 $shell = New-Object -Comobject $comObject
```

Now you want to test for errors by using the *$?* automatic variable. The *$?* automatic variable tells you whether the last command completed successfully. Because *$?* is a Boolean true/false variable, you can use this fact to simplify the coding. You can use the *not* operator, *!*, in conjunction with an *If* statement. If the variable is not true, then you can use the *Throw* statement to raise an error and halt execution of the script. This section of the script is shown here.

```
 if(!$?) { $(Throw "unable to create $comObject object")}
```

If the script successfully creates the *Shell.Application* object, it is now time to provide more feedback as shown here.

```
 Write-Debug "Creating source cab object for $cab"
```

The next step in the operation is to connect to the .cab file by using the *Namespace* method from the *Shell.Application* object as shown here. This is another important step in the process, so it makes sense to use another Write-Debug statement as a progress indicator to the user.

```
 $sourceCab = $shell.Namespace($cab).items()
 Write-Debug "Creating destination folder object for $destination"
```

It is time to connect to the destination folder by using the *Namespace* method as shown here. You also want to use another `Write-Debug` statement to let the user know the folder to which you actually connected.

```
 $DestinationFolder = $shell.Namespace($destination)
 Write-Debug "Expanding $cab to $destination"
```

With all of that preparation out of the way, the actual command that is used to expand the cabinet file is somewhat anticlimactic. You can use the *copyhere* method from the *folder* object that is stored in the *$destinationFolder* variable. You give the reference to the .cab file that is stored in the *$sourceCab* variable as the input parameter as shown here.

```
 $DestinationFolder.CopyHere($sourceCab)
}
```

The starting point to the script accomplishes two things. First, it checks for the presence of the *$debug* variable. If found, it then sets the *$debugPreference* to *continue* to force the `Write-Debug` cmdlet to print messages to the console window. Second, it calls the *ConvertFrom-Cab* function and passes the path to the .cab file from the *–cab* command-line parameter and the destination for the expanded files from the *–destination* parameter as shown here.

```
if($debug) { $debugPreference = "continue" }
ConvertFrom-Cab -cab $cab -destination $destination
```

The completed ExpandCab.ps1 script is shown here.

ExpandCab.ps1

```
Param(
      $cab = "C:\fso\acab.cab",
      $destination = "C:\fso1",
      [switch]$debug
     )
Function ConvertFrom-Cab($cab,$destination)
{
 $comObject = "Shell.Application"
 Write-Debug "Creating $comObject"
 $shell = New-Object -Comobject $comObject
 if(!$?) { $(Throw "unable to create $comObject object")}
 Write-Debug "Creating source cab object for $cab"
 $sourceCab = $shell.Namespace($cab).items()
 Write-Debug "Creating destination folder object for $destination"
 $DestinationFolder = $shell.Namespace($destination)
 Write-Debug "Expanding $cab to $destination"
 $DestinationFolder.CopyHere($sourceCab)
}
```

```
# *** entry point ***
if($debug) { $debugPreference = "continue" }
ConvertFrom-Cab -cab $cab -destination $destination
```

Lack of External Application Support

Many management features still rely on the use of command-line support; a very common example is NETSH. Another example is the MakeCab.exe utility. The *makecab.makecab* COM object was removed from Windows Vista and later versions. To create a .cab file in Windows Vista and beyond, you need to use the MakeCab.exe utility.

First, you need to create a few command-line parameters as shown here.

```
Param(
      $filepath = "C:\fso",
      $path = "C:\fso1\cabfiles",
      [switch]$debug
     )
```

Then you need to create the *New-DDF* function, which creates a basic .ddf file that is used by the MakeCab.exe program to create the .cab file. The syntax for these types of files is documented in the Microsoft Cabinet SDK on MSDN. Once you use the *Function* keyword to create the *New-DDF* function, you can use the `Join-Path` cmdlet to create the file path to the temporary .ddf file you will use. You can concatenate the drive, the folder, and the file name together, but this might become a cumbersome and error-prone operation. As a best practice, you should always use the `Join-Path` cmdlet to build your file paths as shown here.

```
Function New-DDF($path,$filePath)
{
 $ddfFile = Join-Path -path $filePath -childpath temp.ddf
```

It is time to provide some feedback to the user if the script is run with the *–debug* switch by using the `Write-Debug` cmdlet as shown here.

```
 Write-Debug "DDF file path is $ddfFile"
```

You now need to create the first portion of the .ddf file by using an expanding here-string. The advantage of a here-string is that it allows you not to worry about escaping special characters. For example, the comment character in a .ddf file is the semicolon, which is a reserved character in Windows PowerShell. If you try to create the .ddf text without the advantage of using the here-string, you then need to escape each of the semicolons to avoid compile-time errors. By using an expanding here-string, you can take advantage of the expansion of variables. A here-string begins with an at sign and a quotation mark and ends with a quotation mark and an at sign as shown here.

```
 $ddfHeader =@"
;*** MakeCAB Directive file
;
.OPTION EXPLICIT
.Set CabinetNameTemplate=Cab.*.cab
.set DiskDirectory1=C:\fso1\Cabfiles
.Set MaxDiskSize=CDROM
.Set Cabinet=on
.Set Compress=on
"@
```

You may choose to add more feedback for the user via the `Write-Debug` cmdlet as shown here.

```
 Write-Debug "Writing ddf file header to $ddfFile"
```

After providing feedback to the user, you come to the section that might cause some problems. The .ddf file must be a pure ASCII file. By default, Windows PowerShell uses Unicode. To ensure that you have an ASCII file, you must use the `Out-File` cmdlet. You can usually avoid using `Out-File` by using the file redirection arrows; however, this is not one of those occasions. Here is the syntax.

```
 $ddfHeader | Out-File -filepath $ddfFile -force -encoding ASCII
```

You probably want to provide more debug information via the `Write-Debug` cmdlet before you gather your collection of files via the `Get-ChildItem` cmdlet as shown here.

```
 Write-Debug "Generating collection of files from $filePath"
 Get-ChildItem -path $filePath |
```

It is important to filter out folders from the collection because the MakeCab.exe utility is not able to compress folders. To filter folders, use the `Where-Object` cmdlet with a *not* operator stating that the object is not a container as shown here.

```
 Where-Object { !$_.psiscontainer } |
```

After you filter out folders, you need to work with each individual file as it comes across the pipeline by using the `ForEach-Object` cmdlet. Because `ForEach-Object` is a cmdlet as opposed to a language statement, the curly brackets must be on the same line as the `ForEach-Object` cmdlet name. The problem arises in that the curly brackets often get buried within the code. As a best practice, I like to line up the curly brackets unless the command is very short, such as in the previous `Where-Object` command, but this process requires the use of the line continuation character (the backtick). I know some developers who avoid using line continuation, but I personally think that lining up curly brackets is more important because it makes the code easier to read. Here is the beginning of the `ForEach-Object` cmdlet.

```
 Foreach-Object `
```

Because the .dff file used by MakeCab.exe is ASCII text, you need to convert the *FullName* property of the *System.IO.FileInfo* object returned by the `Get-ChildItem` cmdlet to a string. In addition, because you may have files with spaces in their name, it makes sense to ensconce the file *FullName* value in a set of quotation marks as shown here.

```
  {
    '"' + $_.fullname.tostring() + '"'  |
```

You then pipeline the file names to the `Out-File` cmdlet, making sure to specify the ASCII encoding, and use the *–append* switch to avoid overwriting everything else in the text file as shown here.

```
   Out-File -filepath $ddfFile -encoding ASCII -append
  }
```

Now you can provide another update to the debug users and call the *New-Cab* function as shown here.

```
 Write-Debug "ddf file is created. Calling New-Cab function"
 New-Cab($ddfFile)
} #end New-DDF
```

When you enter the *New-Cab* function, you may want to supply some information to the user as shown here.

```
Function New-Cab($ddfFile)
{
 Write-Debug "Entering the New-Cab function. The DDF File is $ddfFile"
```

If the script is run with the *–debug* switch, you can use the */V* parameter of the MakeCab.exe executable to provide detailed debugging information. If the script is not run with the *–debug* switch, you do not want to clutter the screen with too much information and can therefore rely on the default verbosity of the utility as shown here.

```
 if($debug)
    { makecab /f $ddfFile /V3 }
 Else
    { makecab /f $ddfFile }
} #end New-Cab
```

The entry point to the script checks whether the *$debug* variable is present. If it is, the *$debugPreference* automatic variable is set to *continue*, and debugging information is displayed via the `Write-Debug` cmdlet. Once that check is performed, the `New-DDF` cmdlet is called with the *path* and *filepath* values supplied to the command line as shown here.

```
if($debug) {$DebugPreference = "continue"}
New-DDF -path $path -filepath $filepath
```

The completed CreateCab2.ps1 script is shown here.

CreateCab2.ps1

```
Param(
      $filepath = "C:\fso",
      $path = "C:\fso1\cabfiles",
      [switch]$debug
     )
Function New-DDF($path,$filePath)
{
 $ddfFile = Join-Path -path $filePath -childpath temp.ddf
 Write-Debug "DDF file path is $ddfFile"
 $ddfHeader =@"
;*** MakeCAB Directive file
;
.OPTION EXPLICIT
.Set CabinetNameTemplate=Cab.*.cab
.set DiskDirectory1=C:\fso1\Cabfiles
.Set MaxDiskSize=CDROM
.Set Cabinet=on
.Set Compress=on
"@
 Write-Debug "Writing ddf file header to $ddfFile"
 $ddfHeader | Out-File -filepath $ddfFile -force -encoding ASCII
 Write-Debug "Generating collection of files from $filePath"
 Get-ChildItem -path $filePath |
 Where-Object { !$_.psiscontainer } |
 Foreach-Object `
  {
    '"' + $_.fullname.tostring() + '"'  |
   Out-File -filepath $ddfFile -encoding ASCII -append
  }
 Write-Debug "ddf file is created. Calling New-Cab function"
 New-Cab($ddfFile)
} #end New-DDF

Function New-Cab($ddfFile)
{
 Write-Debug "Entering the New-Cab function. The DDF File is $ddfFile"
 if($debug)
    { makecab /f $ddfFile /V3 }
 Else
    { makecab /f $ddfFile }
} #end New-Cab

# *** entry point to script ***
if($debug) {$DebugPreference = "continue"}
New-DDF -path $path -filepath $filepath
```

Additional Resources

- The TechNet Script Center at *http://www.microsoft.com/technet/scriptcenter* has numerous examples of Windows PowerShell scripts that use all of the techniques explored in this chapter.
- Take a look at *Windows PowerShell™ Scripting Guide* (Microsoft Press, 2008) for examples of using WMI and various .NET Framework classes in Windows PowerShell.
- For a good WMI reference, look at *Windows Scripting with WMI Self-Paced Learning Edition* (Microsoft Press, 2006).
- The MSDN reference library has comprehensive product documentation at *http://msdn.microsoft.com/en-us/library/default.aspx* and is the authoritative source for all Microsoft products.
- On the companion media, you will find all of the scripts referred to in this chapter.

CHAPTER 10

Designing Help for Scripts

Although well-written code is easy to understand, easy to maintain, and easy to troubleshoot, it can still benefit from well-written help documentation. Well-written help documentation can list assumptions that were made when the script was written, such as the existence of a particular folder or the need to run as an administrator. It also documents dependencies, such as relying on a particular version of the Microsoft .NET Framework. Good documentation is a sign of a professional at work because it not only informs the user how to get the most from your script, but it also explains how users can modify your script or even use your functions in other scripts.

All production scripts should provide some form of help. But what is the best way to provide that help? In this chapter, you will look at proven methods for providing custom help in Windows PowerShell scripts.

When writing help documentation for a script, three tools are available to you. The first tool is the traditional comment that is placed within the script—the single-line comment that is available in Windows PowerShell 1.0. The second tool is the here-string. The third tool is the multiple-line comment that is introduced in Windows PowerShell 2.0. Once you understand how to use these tools, we will focus on the 13 rules for writing effective comments.

Adding Help Documentation to a Script with Single-Line Comments

Single-line comments are a great way to quickly add documentation to a script. They have the advantage of being simple to use and easy to understand. It is a best practice to provide illuminating information about confusing constructions or to add notes for future work items in the script, and they can be used exclusively within your scripting

environment. In this section, we will look at using single-line comments to add help documentation to a script.

In the CreateFileNameFromDate.ps1 script, the header section of the script uses the comments section to explain how the script works, what it does, and the limitations of the approach. The CreateFileNameFromDate.ps1 script is shown here.

CreateFileNameFromDate.ps1

```
# ------------------------------------------------------------------------
# NAME: CreateFileNameFromDate.ps1
# AUTHOR: ed wilson, Microsoft
# DATE:12/15/2008
#
# KEYWORDS: .NET framework, io.path, get-date
# file, new-item, Standard Date and Time Format Strings
# regular expression, ref, pass by reference
#
# COMMENTS: This script creates an empty text file
# based upon the date-time stamp. Uses format string
# to specify a sortable date. Uses getInvalidFileNameChars
# method to get all the invalid characters that are not allowed
# in a file name. It assumes there is a folder named fso off the
# c:\ drive. If the folder does not exist, the script will fail.
#
# ------------------------------------------------------------------------
Function GetFileName([ref]$fileName)
{
 $invalidChars = [io.path]::GetInvalidFileNamechars()
 $date = Get-Date -format s
 $fileName.value = ($date.ToString() -replace "[$invalidChars]","-") + ".txt"
}

$fileName = $null
GetFileName([ref]$fileName)
new-item -path c:\fso -name $filename -itemtype file
```

In general, you should always provide information on how to use your functions. Each parameter, as well as underlying dependencies, must be explained. In addition to documenting the operation and dependencies of the functions, you should also include information that will be beneficial to those who must maintain the code. You should always assume that the person who maintains your code does not understand what the code actually does, therefore ensuring that the documentation explains everything. In the BackUpFiles.ps1 script, comments are added to both the header and to each function that explain the logic and limitations of the functions as shown here.

BackUpFiles.ps1

```
# ------------------------------------------------------------------------
# NAME: BackUpFiles.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 12/12/2008
#
# KEYWORDS: Filesystem, get-childitem, where-object
# date manipulation, regular expressions
#
# COMMENTS: This script backs up a folder. It will
# back up files that have been modified within the past
# 24 hours. You can change the interval, the destination,
# and the source. It creates a backup folder that is named based upon
# the time the script runs. If the destination folder does not exist, it
# will be created. The destination folder is based upon the time the
# script is run and will look like this: C:\bu\12.12.2008.1.22.51.PM.
# The interval is the age in days of the files to be copied.
#
# ------------------------------------------------------------------------
Function New-BackUpFolder($destinationFolder)
{
 #Receives the path to the destination folder and creates the path to
 #a child folder based upon the date / time. It then calls the New-Backup
 #function while passing the source path, destination path, and interval
 #in days.
 $dte = get-date
 #The following regular expression pattern removes white space, colon,
 #and forward slash from the date and replaces with a period to create the
 #backup folder name.
 $dte = $dte.tostring() -replace "[:\s/]", "."
 $backUpPath = "$destinationFolder" + $dte
 $null = New-Item -path $backUpPath -itemType directory
 New-Backup $dataFolder $backUpPath $backUpInterval
} #end New-BackUpFolder

Function New-Backup($dataFolder,$backUpPath,$backUpInterval)
{
 #Does a recursive copy of all files in the data folder and filters out
 #all files that have been written to within the number of days specified
 #by the interval. Writes copied files to the destination and will create
 #if the destination (including parent path) does not exist. Will overwrite
 #if destination already exists. This is unlikely, however, unless the
 #script is run twice during the same minute.
 "backing up $dataFolder... check $backUppath for your files"
 Get-Childitem -path $dataFolder -recurse |
 Where-Object { $_.LastWriteTime -ge (get-date).addDays(-$backUpInterval) } |
```

```
 Foreach-Object { copy-item -path $_.FullName -destination $backUpPath -force }
} #end New-BackUp

# *** entry point to script ***

$backUpInterval = 1
$dataFolder = "C:\fso"
$destinationFolder = "C:\BU\"
New-BackupFolder $destinationFolder
```

NOTES FROM THE FIELD

Crafting Inspired cmdlet Help

Dean Tsaltas, Microsoft Scripting Guy Emeritus

In many ways, writing cmdlet help is no different from writing any other type of help documentation. If you want to do a really good job, you must "become your user." This is easier said than done, of course—especially if you are the person who designed and implemented the cmdlets for which you are writing the help. Even though you just created the cmdlets, you can only guess at the mysterious ways in which some of your users will use and abuse your creations. That said, you must give it your all. Rent the original *Karate Kid* and watch it for inspiration. Wax on and wax off before hitting the keyboard. After crafting just the right sentences to convey a concept, remember to ask yourself, "What ambiguity is left in what I just wrote? What can my user possibly still question after reading my text?" Picture yourself explaining the concept to your users, anticipate their questions, and answer them.

For example, suppose that your cmdlet creates some type of file and takes a name or a full path that includes a name as a parameter. Anticipate the questions that users will have about that parameter: how long can it be, are any characters disallowed, how are quotes within quotes handled, will the resultant file include an extension or should I include the appropriate extension in the parameter value? Don't force your users to experiment to answer questions that you can easily anticipate and to which you can quickly provide answers. Help them.

Next, remember that a single example is worth a thousand support calls. You should aim high when it comes to examples. It is a best practice to brainstorm the top tasks that you think your users will be trying to accomplish. At a minimum, you need to include an example for each of those top tasks. Once you have established that baseline, you should aim to provide an example that exercises each and every cmdlet parameter set. Even if you simply mine your test cases for bland examples, try to provide your users with a starting point. As you well know, it's much easier

to manipulate a working command line and get it to do what you want than it is to start from scratch.

It's important to consider how your users will interact with cmdlet help. They will see it at a command prompt one full screen at a time. Because that first screen is like the "above-the-fold" section of a newspaper, make sure you handle any really important issues right there in the detailed description. If you need certain privileges to use the cmdlet, let your users know that information up front. If there's an associated provider that might be useful to them, tell your users about it early.

Don't neglect the Related Links section of your help. It's very easy to simply list all of the cmdlets with the same noun, especially when you're in a rush. Yet, are those truly the only cmdlets that are related to the one you're writing about? For instance, is there another cmdlet that your users must use to generate an object that your cmdlet can accept as a parameter value? If so, this other cmdlet also deserves a place in the Related Links list. Again, imagine having a discussion with your users. What other help can you suggest that they access? Also include links to this additional help and not just to the help that is obviously related based on cmdlet naming conventions.

My last bit of advice about writing cmdlet help is to write it as early as you can in the development cycle and get it in the hands of some pre-alpha users to start the feedback cycle quickly. The only way to develop excellent cmdlet help (or any other type of technical documentation) is through iterative improvements in response to feedback. Include numerous simple examples in the help as soon as you can. Having someone use a cmdlet with no accompanying help is unlikely to help you understand what information is needed by your users to get the job done. However, providing someone with three examples will certainly elicit a user response as to what the fourth and fifth examples should be.

Using the Here-String for Multiple-Line Comments

One method that can be used in Windows PowerShell 1.0 to allow for multiline comments is to use a here-string. The here-string allows you to assign text without worrying about line formatting or escaping quotation marks and other special characters. It is helpful when working with multiple lines of text because it allows you to overlook the more tedious aspects of working with formatted text, such as escaping quotation marks. The advantage of using a here-string to store your comments is that it then becomes rather easy to use another script to retrieve all of the comments.

Constructing a Here-String

An example of working with here-strings is the Demo-HereString.ps1 script. The *$instructions* variable is used to hold the content of the here-string. The actual here-string itself is created by beginning the string with the at and double quotation mark (@") symbol. The here-string is terminated by reversing the order with the double quotation mark and the at symbol ("@). Everything between the two tags is interpreted as a string, including special characters. The here-string from the Demo-HereString.ps1 script is shown here.

```
$instructions = @"
This command line demo illustrates working with multiple lines
of text. The cool thing about using a here-string is that it allows
you to "work" with text without the need to "worry" about quoting
or other formating issues.
   It even allows you
     a sort of
       wysiwyg type of experience.
You format the data as you wish it to appear.
"@
```

TRADEOFF

Multiple-Line Comments in Windows PowerShell 1.0

If you want to include a comment in Windows PowerShell 1.0 that spans multiple lines, you can use multiple pound characters (#) as shown here.

```
# This is the first line of a multiple line comment
# This is the second line of the comment
```

This process works fine for short comments, but when you need to write a paragraph documenting a particular feature or construction in the script, this method becomes rather annoying because of the need to type all of the pound characters. If one of the comment characters is inadvertently omitted, an error is generated. Depending on the actual placement of the pound sign, the error can be misleading and cause you to waste development time by chasing down the errant line of code. In Windows PowerShell 2.0, you can still use the multiple pound character approach to adding comments if desired. The advantages to doing so are simplicity and backward compatibility with Windows PowerShell 1.0. Your code will also be easy to read because anyone familiar with Windows PowerShell will immediately recognize the lines of code as comments.

The here-string is displayed by calling the variable that contains the here-string. In the Demo-HereString.ps1 script, the response to a prompt posed by the `Read-Host` cmdlet is stored in the *$response* variable. The `Read-Host` command is shown here.

```
$response = Read-Host -Prompt "Do you need instructions? <y / n>"
```

The value stored in the *$response* variable is then evaluated by the *If* statement. If the value is equal to the letter "y," the contents of the here-string are displayed. If the value of the *$response* variable is equal to anything else that the script displays, the string displays "good bye" and exits. This section of the script is shown here.

```
if ($response -eq "y") { $instructions ; exit }
else { "good bye" ; exit }
```

The Demo-HereString.ps1 script is seen here.

Demo-HereString.ps1

```
$instructions = @"
This command line demo illustrates working with multiple lines
of text. The cool thing about using a here-string is that it allows
you to "work" with text without the need to "worry" about quoting
or other formating issues.
   It even allows you
     a sort of
       wysiwyg type of experience.
You format the data as you wish it to appear.
"@

$response = Read-Host -Prompt "Do you need instructions? <y / n>"
if ($response -eq "y") { $instructions ; exit }
else { "good bye" ; exit }
```

An Example: Adding Comments to a Registry Script

To better demonstrate the advantages of working with here-strings, consider the following example that employs the registry. A script named GetSetieStartPage.ps1 reads or modifies a few values from the registry to configure the Internet Explorer start pages. The GetSetieStartPage.ps1 script contains pertinent comments and provides a good example for working with documentation.

With Internet Explorer 7.0, there are actually two registry keys that govern the Internet Explorer start page. The registry key that is documented in the Tweakomatic program, available from the Microsoft Script Center, accepts a single string for the start page, which makes sense because traditionally you can have only a single start page. For Internet Explorer 7, an additional registry key was added to accept multiple strings (an array of strings), which in turn gives you the ability to have multiple start pages. You can use the Windows Management

Instrumentation (WMI) *stdRegProv* class to both read and edit the registry keys. The main advantage of this technique is that it gives you the ability to edit the registry remotely.

The registry keys that are involved with the settings are shown in Figure 10-1.

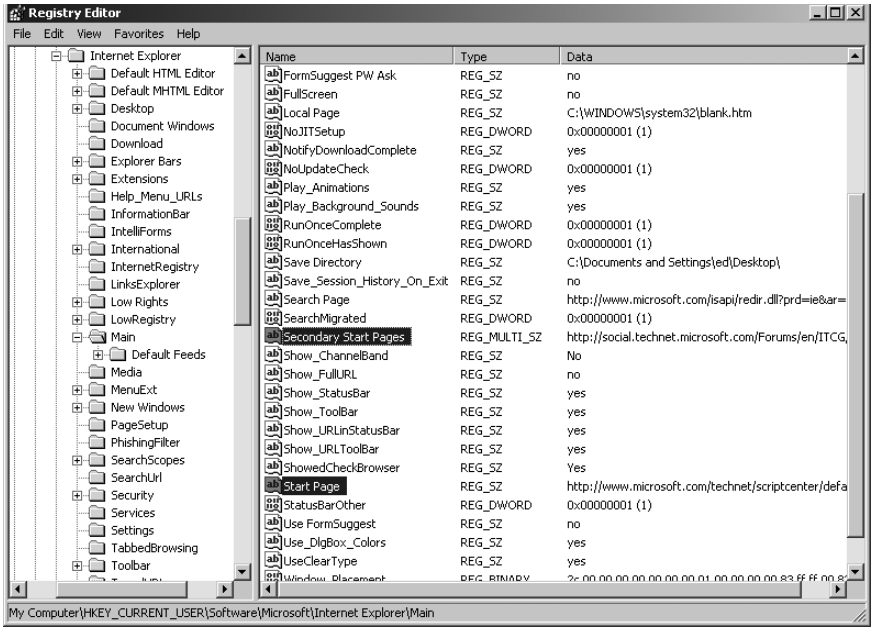


FIGURE 10-1 Internet Explorer registry keys shown in the Registry Editor

First, you must create a few command-line parameters. Two of these are switched parameters, which allow you to control the way the script operates—either by obtaining information from the registry or setting information in the registry. The last parameter is a regular parameter that controls the target computer. You are assigning a default value for the *$computer* variable of localhost, which means that the script reads the local registry by default as shown here.

```
Param([switch]$get,[switch]$set,$computer="localhost")
```

You now need to create the *Get-ieStartPage* function. The *Get-ieStartPage* function will be used to retrieve the current start page settings. To create a function, you can use the *Function* keyword and give it a name as shown here.

```
Function Get-ieStartPage()
```

After using the *Function* keyword to create the new function, you need to add some code to the script block. First, you must create the variables to be used in the script. These are the same types of variables that you use when using WMI to read from the registry in Microsoft

VBScript. You must specify the registry hive from which you plan on querying by using one of enumeration values shown Table 10-1.

TABLE 10-1 WMI Registry Tree Values

NAME	VALUE
HKEY_CLASSES_ROOT	2147483648
HKEY_CURRENT_USER	2147483649
HKEY_LOCAL_MACHINE	2147483650
HKEY_USERS	2147483651
HKEY_CURRENT_CONFIG	2147483653
HKEY_DYN_DATA	2147483654

In VBScript, you often create a constant to hold the WMI registry tree values (although a regular variable is fine if you do not change it). If you feel that you must have a constant, the following code is the syntax you need to use.

```
New-Variable -Name hkcu -Value 2147483649 -Option constant
```

The *$key* variable is used to hold the path to the registry key with which you will be working. The *$property* and *$property2* variables are used to hold the actual properties that will control the start pages. This section of the script is shown here.

```
{
 $hkcu = 2147483649
 $key = "Software\Microsoft\Internet Explorer\Main"
 $property = "Start Page"
 $property2 = "Secondary Start Pages"
```

You now need to use the [WMICLASS] type accelerator to create an instance of the *stdRegProv* WMI class. You can hold the management object that is returned by the type accelerator in the *$wmi* variable. What is a bit interesting is the path to the WMI class. It includes both the computer name, the WMI namespace followed by a colon, and the name of the WMI class. This syntax corresponds exactly to the *Path* property that is present on all WMI classes (because it is inherited from the *System* abstract class) as shown in Figure 10-2. (If you are interested in this level of detail about WMI, you can refer to *Windows Scripting with WMI: Self-Paced Learning Guide* [Microsoft Press, 2006]. All of the examples are written in VBScript, but the book applies nearly 100 percent to Windows PowerShell.)

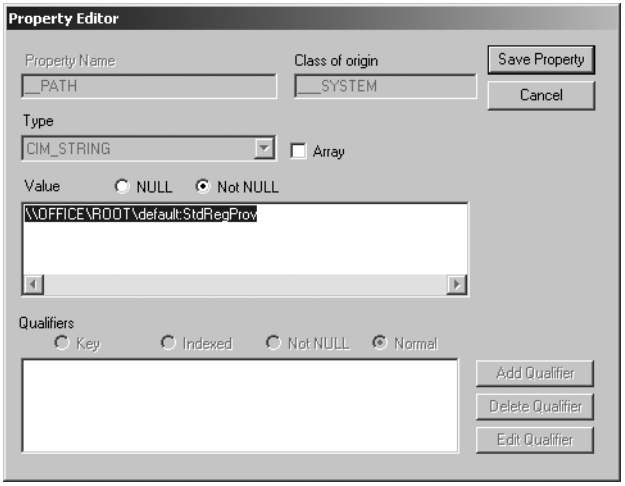


FIGURE 10-2 The WMI *Path* property seen in the WbemTest utility

Here is the line of code that creates an instance of the *stdRegProv* WMI class on the target computer.

```
$wmi = [wmiclass]"\\$computer\root\default:stdRegProv"
```

Next, you need to use the *GetStringValue* method because you want to obtain the value of a string (it really is that simple). This step can be a bit confusing when using the *stdRegProv* WMI class because of the large number of methods in this class. For each data type you want to access, you must use a different method for both writing and reading from the registry. This also means that you must know what data type is contained in the registry key property value with which you want to work. The *stdRegProv* methods are documented in Table 10-2.

TABLE 10-2 *stdRegProv* Methods

NAME	DEFINITION
CheckAccess	System.Management.ManagementBaseObject CheckAccess(System.UInt32 hDefKey, System.String sSubKeyName, System.UInt32 uRequired)
CreateKey	System.Management.ManagementBaseObject CreateKey(System.UInt32 hDefKey, System.String sSubKeyName)
DeleteKey	System.Management.ManagementBaseObject DeleteKey(System.UInt32 hDefKey, System.String sSubKeyName)
DeleteValue	System.Management.ManagementBaseObject DeleteValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName)

NAME	DEFINITION
EnumKey	System.Management.ManagementBaseObject EnumKey (System.UInt32 hDefKey, System.String sSubKeyName)
EnumValues	System.Management.ManagementBaseObject EnumValues(System.UInt32 hDefKey, System.String sSubKeyName)
GetBinaryValue	System.Management.ManagementBaseObject GetBinaryValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName)
GetDWORDValue	System.Management.ManagementBaseObject GetDWORDValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName)
GetExpandedStringValue	System.Management.ManagementBaseObject GetExpandedStringValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName)
GetMultiStringValue	System.Management.ManagementBaseObject GetMultiStringValue(System.UInt32 hDefKey, System.StringsSubKeyName, System.String sValueName)
GetStringValue	System.Management.ManagementBaseObject GetStringValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName)
SetBinaryValue	System.Management.ManagementBaseObject SetBinaryValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName, System.Byte[] uValue)
SetDWORDValue	System.Management.ManagementBaseObject SetDWORDValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName, System.UInt32 uValue)
SetExpandedStringValue	System.Management.ManagementBaseObject SetExpandedStringValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName, System.String sValue)
SetMultiStringValue	System.Management.ManagementBaseObject SetMultiStringValue(System.UInt32 hDefKey, System.StringsSubKeyName, System.String sValueName, System.String[] sValue)
SetStringValue	System.Management.ManagementBaseObject SetStringValue(System.UInt32 hDefKey, System.String sSubKeyName, System.String sValueName, System.String sValue)

The code that obtains the value of the default Internet Explorer home page is shown here.

```
($wmi.GetStringValue($hkcu,$key,$property)).sValue
```

After obtaining the value of the string that holds the default home page, you need to obtain the value of a multistring registry key that is used for the additional home pages. To do this, you can use the *GetMultiStringValue* method. What is convenient about this method is that the values of the array that are returned are automatically expanded, and you can thus avoid the for … next gyrations required when performing this method call when using VBScript. This line of code is shown here.

```
($wmi.GetMultiStringValue($hkcu,$key, $property2)).sValue
```

Adding comments to the closing curly brackets is a best practice that enables you to quickly know where the function begins and ends.

LESSONS LEARNED

Pairing a Comment with a Closing Curly Bracket

I once spent an entire train ride in Germany that went from Regensburg to Hamburg (nearly a five-hour trip) troubleshooting a problem with a script that occurred as the train left the central train station in Regensburg. The script was to be used for the *Windows Vista Resource Kit* (Microsoft Press, 2008), and I had a deadline to meet. The problem occurred with an edit that I made to the original script, and I forgot to close the curly bracket. The error was particularly misleading because it pointed to a line in the very long script that was unrelated to the issue at hand. It was on this train ride that I learned the value of adding a comment to closing curly brackets, which is now something that I nearly always do.

Here is the closing curly bracket and associated comment. If you always type comments in the same pattern (for example, #end with no space), they are then easy to spot if you ever decide to write a script to search for them.

```
} #end Get-ieStartPage
```

You now need to create a function to assign new values to the Internet Explorer start pages. You can call the *Set-ieStartPage* function as shown here.

```
Function Set-ieStartPage()
{
```

You must assign some values to a large number of variables. The first four variables are the same ones used in the previous function. (You could have made them script-level variables and saved four lines of code in the overall script, but then the functions would not have been stand-alone pieces of code.) The *$value* variable is used to hold the default home page, and the *$aryvalues* variable holds an array of secondary home page URLs. This section of the code is shown here.

```
$hkcu = 2147483649
$key = "Software\Microsoft\Internet Explorer\Main"
$property = "Start Page"
$property2 = "Secondary Start Pages"
$value = "http://www.microsoft.com/technet/scriptcenter/default.mspx"
$aryValues = "http://social.technet.microsoft.com/Forums/en/ITCG/threads/",
"http://www.microsoft.com/technet/scriptcenter/resources/qanda/all.mspx"
```

After assigning values to variables, you can use the [WMICLASS] type accelerator to create an instance of the *stdRegProv* WMI class. This same line of code is used in the *Get-ieStartPage* function and is shown here.

```
$wmi = [wmiclass]"\\$computer\root\default:stdRegProv"
```

You can now use the *SetStringValue* method to set the value of the string. The *SetStringValue* method takes four values. The first is the numeric value representing the registry hive to which to connect. The next is the string for the registry key. The third position holds the property to modify, and last is a string representing the new value to assign as shown here.

```
$rtn = $wmi.SetStringValue($hkcu,$key,$property,$value)
```

Next, you can use the *SetMultiStringValue* method to set the value of a multistring registry key. This method takes an array in the fourth position. The signature of the *SetMultiStringValue* method is similar to the *SetStringValue* signature. The only difference is that the fourth position needs an array of strings and not a single value as shown here.

```
$rtn2 = $wmi.SetMultiStringValue($hkcu,$key,$property2,$aryValues)
```

Now, you can print the value of the *ReturnValue* property. The *ReturnValue* property contains the error code from the method call. A zero means that the method worked (no runs, no errors), and anything else means that there was a problem as shown here.

```
  "Setting $property returned $($rtn.returnvalue)"
  "Setting $property2 returned $($rtn2.returnvalue)"
} #end Set-ieStartPage
```

You are now at the entry point to the script. You must first get the starting values and then set them to the new values that you want to configure. If you want to re-query the registry to ensure that the values took effect, you can simply call the *Get-ieStartPage* function again as shown here.

```
if($get) {Get-ieStartpage}
if($set){Set-ieStartPage}
```

The complete GetSetieStartPage.ps1 script is shown here.

GetSetieStartPage.ps1

```
Param([switch]$get,[switch]$set,$computer="localhost")
$Comment = @"
NAME: GetSetieStartPage.ps1
AUTHOR: ed wilson, Microsoft
DATE: 1/5/2009

KEYWORDS: stdregprov, ie, [wmiclass] type accelerator,
Hey Scripting Guy
COMMENTS: This script uses the [wmiclass] type accelerator
and the stdregprov to get the ie start pages and to set the
ie start pages. Using ie 7 or better you can have multiple
start pages.

"@ #end comment

Function Get-ieStartPage()
{
$Comment = @"
FUNCTION: Get-ieStartPage
Is used to retrieve the current settings for Internet Explorer 7 and greater.
The value of $hkcu is set to a constant value from the SDK that points
to the Hkey_Current_User. Two methods are used to read
from the registry because the start page is single valued and
the second start page's key is multi-valued.

"@ #end comment
 $hkcu = 2147483649
 $key = "Software\Microsoft\Internet Explorer\Main"
 $property = "Start Page"
 $property2 = "Secondary Start Pages"
 $wmi = [wmiclass]"\\$computer\root\default:stdRegProv"
 ($wmi.GetStringValue($hkcu,$key,$property)).sValue
 ($wmi.GetMultiStringValue($hkcu,$key, $property2)).sValue
} #end Get-ieStartPage

Function Set-ieStartPage()
{
$Comment = @"
FUNCTION: Set-ieStartPage
Allows you to configure one or more home pages for IE 7 and greater.
The $aryValues and the $Value variables hold the various home pages.
Specify the complete URL ex: "http://www.ScriptingGuys.Com." Make sure
to include the quotation marks around each URL.

"@ #end comment
```

```
  $hkcu = 2147483649
  $key = "Software\Microsoft\Internet Explorer\Main"
  $property = "Start Page"
  $property2 = "Secondary Start Pages"
  $value = "http://www.microsoft.com/technet/scriptcenter/default.mspx"
  $aryValues = "http://social.technet.microsoft.com/Forums/en/ITCG/threads/",
  "http://www.microsoft.com/technet/scriptcenter/resources/qanda/all.mspx"
  $wmi = [wmiclass]"\\$computer\root\default:stdRegProv"
  $rtn = $wmi.SetStringValue($hkcu,$key,$property,$value)
  $rtn2 = $wmi.SetMultiStringValue($hkcu,$key,$property2,$aryValues)
  "Setting $property returned $($rtn.returnvalue)"
  "Setting $property2 returned $($rtn2.returnvalue)"
} #end Set-ieStartPage

# *** entry point to script
if($get) {Get-ieStartpage}
if($set){Set-ieStartPage}
```

Retrieving Comments by Using Here-Strings

Due to the stylized nature of here-strings, you can use a script, such as GetCommentsFromScript.ps1, to retrieve the here-strings from another script, such as GetSetieStartPage.ps1. You are interested in obtaining three comment blocks. The first comment block contains normal script header information: title, author, date, keywords, and comments on the script itself. The second and third comment blocks are specifically related to the two main functions contained in the GetSetieStartPage.ps1 script. When processed by the GetCommentsFromScript.ps1 script, the result is automatically produced script documentation. To write comments from the source file to another document, you need to open the original script, search for your comments, and write the appropriate text to a new file.

The GetCommentsFromScript.ps1 script begins with the *Param* statement. The *Param* statement is used to allow you to provide information to the script at run time. The advantage of using a command-line parameter is that you do not need to open the script and edit it to provide the path to the script whose comments you are going to copy. You are making this parameter a mandatory parameter by assigning a default value to the *$script* variable. The default value you assign uses the *Throw* statement to raise an error, which means that the script will always raise an error when run unless you supply a value for the *–script* parameter when you run the script.

Using *Throw* to Raise an Error

Use of the *Throw* statement is seen in the DemoThrow.ps1 script. To get past the error that is raised by the *Throw* statement in the *Set-Error* function, you first need to set the value of the *$errorActionPreference* variable to SilentlyContinue, which causes the error to not be

displayed and allows the script to continue past the error. (This variable performs the same action as the On Error Resume Next setting from VBScript.) The *If* statement is used to evaluate the value of the *$value* variable. If there is a match, the *Throw* statement is encountered and the exception is thrown.

To evaluate the error, you can use the *Get-ErrorDetails* function. The error count is displayed first, and it will be incremented by one due to the error that was raised by the *Throw* statement. You can then take the first error (the error with the index value of 0 is always the most recent error that occurred) and send the error object to the `Format-List` cmdlet. You choose all of the properties. However, the invocation information is returned as an object. Therefore, you must query that object directly by accessing the invocation object via the *Invocationinfo* property of the error object. The resulting error information is shown in Figure 10-3.

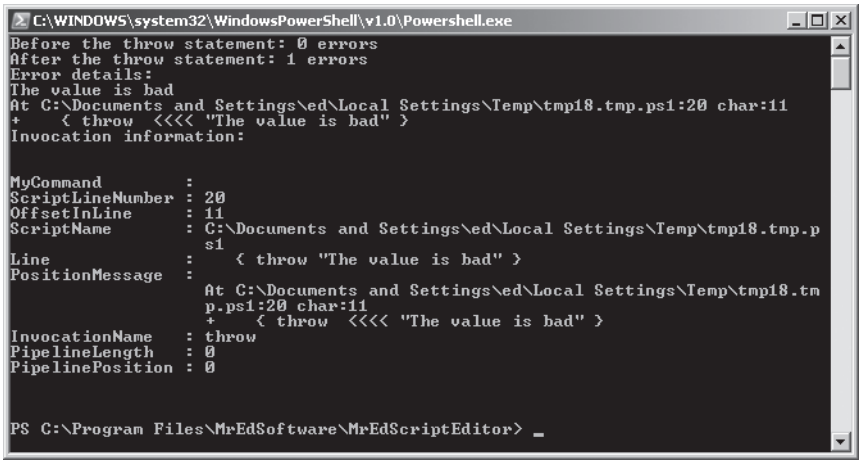


FIGURE 10-3 The *Throw* statement is used to raise an error.

The complete DemoThrow.ps1 script is shown here.

DemoThrow.ps1

```
Function Set-Error
{
 $errorActionPreference = "SilentlyContinue"
 "Before the throw statement: $($error.count) errors"
 $value = "bad"
 If ($value -eq "bad")
   { throw "The value is bad" }
} #end Set-Error

Function Get-ErrorDetails
{
 "After the throw statement: $($error.count) errors"
 "Error details:"
 $error[0] | Format-List -Property *
```

```
 "Invocation information:"
 $error[0].InvocationInfo
} #end Get-ErrorDetails

# *** Entry Point to Script
Set-Error
Get-ErrorDetails
```

The *Param* statement is shown here.

```
Param($Script= $(throw "The path to a script is required."))
```

You need to create a function that creates a file name for the new text document that will be created as a result of gleaning all of the comments from the script. To create the function, you can use the *Function* keyword and follow it with the name for the function. In your case, you can call the function *Get-FileName* in keeping with the spirit of the verb-noun naming convention in Windows PowerShell. The function will take a single input parameter that is held in the *$script* variable inside the function. The *$script* variable will hold the path to the script to be analyzed. The entry to the *Get-FileName* function is shown here.

```
Function Get-FileName($Script)
{
```

Working with Temporary Folders

Next, you can obtain the path to the temporary folder on the local computer in many different ways, including using the environmental PS drive. This example uses the static *GetTempPath* method from the *System.Io.Path* .NET Framework class. The *GetTempPath* method returns the path to the temporary folder, which is where you will store the newly created text file. You hold the temporary folder path in the *$outputPath* variable as shown here.

```
 $outputPath = [io.path]::GetTempPath()
```

You decide to name your new text file after the name of the script. To do this, you need to separate the script name from the path in which the script is stored. You can use the *Split-Path* function to perform this surgery. The *–leaf* parameter instructs the cmdlet to return the script name. If you want the directory path that contains the script, you can use the *–parent* parameter. You put the `Split-Path` cmdlet inside a pair of parentheses because you want that operation to occur first. When the dollar sign is placed in front of the parentheses, it creates a subexpression that executes the code and then returns the name of the script. You can use .ps1 as the extension for your text file, but that can become a bit confusing because it is the extension for a script. Therefore, you can simply add a .txt extension to the returned file name and place the entire string within a pair of quotation marks.

You can use the Join-Path cmdlet to create a new path to your output file. The new path is composed of the temporary folder that is stored in the *$outputPath* variable and the file name you created using *Split-Path*. You combine these elements by using the Join-Path cmdlet. You can use string manipulation and concatenation to create the new file path, but it is much more reliable to use the Join-Path and Split-Path cmdlets to perform these types of operations. This section of the code is shown here.

```
 Join-Path -path $outputPath -child "$(Split-Path $script -leaf).txt"
} #end Get-FileName
```

You need to decide how to handle duplicate files. You can prompt the user by saying that a duplicate file exists, which looks like the code shown here.

```
      $Response = Read-Host -Prompt "$outputFile already exists. Do you wish to delete
it <y / n>?"
      if($Response -eq "y")
        { Remove-Item $outputFile | Out-Null }
     ELSE { "Exiting now." ; exit }
```

You can implement some type of naming algorithm that makes a backup of the duplicate file by renaming it with an .old extension, which looks like the code shown here.

```
          if(Test-Path -path "$outputFile.old") { Remove-Item -Path "$outputFile.old" }
          Rename-Item -path $outputFile -newname  "$(Split-Path $outputFile -leaf).old"
```

You can also simply delete the previously existing file, which is what I generally choose to do. The action you want to perform goes into the *Remove-OutPutFile* function. You begin the function by using the *Function* keyword, specifying the name of the function, and using the *$outputFile* variable for input to the function as shown here.

```
Function Remove-outputFile($outputFile)
{
```

To determine whether the file exists, you can use the Test-Path cmdlet and supply the string contained in the *$outputFile* variable to the *–path* parameter. The Test-Path cmdlet only returns a *true* or *false* value. When a file is not found, it returns a *false* value, which means that you can use the *If* statement to evaluate the existence of the file. If the file is found, you can perform the action in the script block. If the file is not found, the script block is not executed. As shown here, the first command does not find the file, and *false* is returned. In the second command, the script block is not executed because the file cannot be located.

```
PS C:\> Test-Path c:\missingfile.txt
False
PS C:\> if(Test-Path c:\missingfile.txt){"found file"}
PS C:\>
```

Inside the *Remove-OutPutFile* function, you can use the *If* statement to determine whether the file referenced by *$outputFile* already exists. If it does, it is deleted by using the Remove-Item

cmdlet. The information that is normally returned when a file is deleted is pipelined to the `Out-Null` cmdlet providing for a silent operation. This portion of the code is shown here.

```
  if(Test-Path -path $outputFile) { Remove-Item $outputFile | Out-Null }

} #end Remove-outputFile
```

After you create the name for the output file and delete any previous output files that might be around, it is time to retrieve the comments from the script. To do this, you can create the *Get-Comments* function and pass it both the *$script* variable and *$outputFile* variable as shown here.

```
Function Get-Comments($Script,$outputFile)
{
```

Reading the Comments in the Output File

It is now time to read the text of the script. You can use the `Get-Content` cmdlet and provide it with the path to the script. When you use `Get-Content` to read a file, the file is read one line at a time and passed along the pipeline. If you store the result into a variable, you will have an array. You can treat the *$a* variable as any other array, including obtaining the number of elements in the array via the *Length* property and indexing directly into the array as shown here.

```
PS C:\fso> $a = Get-Content -Path C:\fso\GetSetieStartPage.ps1
PS C:\fso> $a.Length
62
PS C:\fso> $a[32]
($wmi.GetMultiStringValue($hkcu,$key, $property2)).sValue
```

The section of the script that reads the input script and sends it along the pipeline is shown here.

```
 Get-Content -path $Script |
```

Next, you need to look inside each line to determine whether it belongs to the comment block. To examine each line within a pipeline, you must use the `ForEach-Object` cmdlet. This cmdlet is similar to a *Foreach … next* statement in that it lets you work with an individual object from within a collection one at a time. The backtick character (`) is used to continue the command to the next line. The action you want to perform on each object as it comes across the pipeline is contained inside a script block that is delineated with a set of curly brackets (braces). This part of the *Get-Content* function is shown here.

```
 Foreach-Object `
  {
```

Once you are inside the `ForEach-Object` cmdlet process block, it is time to examine the line of text. To do this, you can use the *If* statement. The *$_* automatic variable is used to represent the current line that is on the pipeline. You use the *–match* operator to perform

a regular expression pattern match against the line of text. The *–match* operator returns a Boolean value—either *true* or *false*—in response to the pattern as shown here.

```
PS C:\fso> '$Comment = @"' -match  '^\$comment\s?=\s?@"'
True
```

The regular expression pattern you are using is composed of a number of special characters as shown in Table 10-3.

TABLE 10-3 Regular Expression Match Pattern and Meaning

CHARACTER	DESCRIPTION
^	Match at the beginning
\	Escape character so the $ sign is treated as a literal character and not the special character used in regular expressions
$comment	Literal characters
\s?	Zero or more white space characters
=	Literal character
@"	Literal characters

The section of code that examines the current line of text on the pipeline is shown here.

```
If($_ -match '^\$comment\s?=\s?@"')
```

You can create a variable named *$beginComment* that is used to mark the beginning of the comment block. If you make it past the *–match* operator, you find the beginning of the comment block. You can set the variable equal to *$true* as shown here.

```
{
 $beginComment = $True
} #end if match @"
```

Next, you can see whether you are at the end of the comment block by once again using the *–match* operator. You will look for the @" character sequence that is used to close a here-string. If you find this sequence, you can set the *$beginComment* variable to false as shown here.

```
If($_ -match '"@')
  {
   $beginComment = $False
  } #end if match "@
```

After you pass the first two *If* statements—the first identifying the beginning of the here-string and the second locating the end of the here-string—you now want to grab the text that needs to be written to your comment file by setting the *$beginComment* variable to true. You also want to ensure that you do not see the @" character on the line because this designates the end of the here-string. To make this determination, you can use a compound *If* statement as shown here.

```
If($beginComment -AND $_ -notmatch '@"')
  {
```

It is now time to write the text to the output file. To do this, you can use the *$_* automatic variable, which represents the current line of text, and pipeline it to the `Out-File` cmdlet. The `Out-File` cmdlet receives the *$outputFile* variable that contains the path to the comment file. You can use the *–append* parameter to specify that you want to gather all of the comments from the script into the comment file. If you do not use the *–append* parameter, the text file will only contain the last comment because, by default, the `Out-File` cmdlet will overwrite the contents of any previously existing file. You can then add closing curly brackets for each of the comments that were previously opened. I consider it a best practice to add a comment after each closing curly bracket that indicates the purpose of the brace. This procedure makes the script much easier to read, troubleshoot, and maintain. This section of the code is shown here.

```
     $_ | Out-File -FilePath $outputFile -append
    } # end if beginComment
  } #end Foreach
} #end Get-Comments
```

You can now create a function named *Get-OutPutFile* that opens the output file for you to read. Because the temporary folder is not easy to find and because you have the path to the file in the *$outputFile* variable, it makes sense to use the script to open the output file. The *Get-OutPutFile* function receives a single input variable named *$outputFile*. When you call the *Get-OutPutFile* function, you pass a variable to the function that contains the path to the comment file that you want to open. That path is contained in the *$outputFile* variable. You can pass any value to the *Get-OutPutFile* function. Once inside the function, the value is then referred to by the *$outputFile* variable. You can even pass a string directly to the function without even using quotation marks around the string as shown here.

```
Function Get-outputFile($outputFile)
{
 Notepad $outputFile
} #end Get-outputFile

Get-outputFile -outputfile C:\fso\GetSetieStartPage.ps1
```

LESSONS LEARNED

Don't Mess with the Worker Section of the Script

If I am going to gather data to pass to a function when writing a script, I generally like to encase the data in the same variable name that will be used both outside and inside the function. One reason for doing this is because it follows one of my best practices for script development: "Don't mess with the worker section of the script." In the *Get-OutPutFile* function, you are "doing work." To change the function in future scripts requires that you edit the string literal value, whereby you run the risk of breaking the code because many methods have complicated constructors. If you are also trying to pass values to the method constructors that require escaping special characters, then the risk of making a mistake becomes even worse.

By placing the string in a variable, you can easily edit the value of the variable. In fact, you are set up to provide the value of the variable via the command line or to base the value on an action performed in another function. Whenever possible, you should avoid placing string literal values directly in the script. In the code that follows, you can use a variable to hold the path to the file that is passed to the *Get-OutPutFile* function.

```
Function Get-outputFile($outputFile)
{
 Notepad $outputFile
} #end Get-outputFile

$outputFile = "C:\fso\GetSetieStartPage.ps1"
Get-outputFile -outputfile $outputFile
```

The complete *Get-OutPutFile* function is shown here.

```
Function Get-outputFile($outputFile)
{
 Notepad $outputFile
} #end Get-outputFile
```

Instead of typing in a string literal value for the path to the output file, the *$outputFile* variable receives the path that is created by the *Get-FileName* function. The *Get-FileName* function receives the path to the script that contains the comments to be extracted. The path to this script comes in via the command-line parameter. When a function has a single input parameter, you can pass it to the function by using a set of smooth parentheses. On the other hand, if the function uses two or more input parameters, you must use the *–parameter* name syntax. This line of code is shown here.

```
$outputFile = Get-FileName($script)
```

Next, you can call the *Remove-OutPutFile* function and pass it the path to the output file that is contained in the *$outputFile* variable. The *Remove-OutPutFile* function was discussed in the "Working with Temporary Folders" section earlier in this chapter. This line of code is shown here.

```
Remove-outputFile($outputFile)
```

Once you are assured of the name of your output file, you can call the *Get-Comments* function to retrieve comments from the script whose path is indicated by the *$script* variable. The comments are written to the output file referenced by the *$outputFile* variable as shown here.

```
Get-Comments -script $script -outputfile $outputFile
```

When all of the comments are written to the output file, you can finally call the *Get-OutPutFile* function and pass it the path contained in the *$outputFile* variable. If you do not want the comment file to be opened, you can easily comment the line out of your script or you can delete it and the *Get-OutPutFile* function itself from your script. If you are interested in reviewing each file prior to saving it, leave the line of code in place. This section of the script is shown here.

```
Get-outputFile($outputFile)
```

When the GetCommentsFromScript.ps1 script runs, nothing is emitted to the console. The only confirmation message that the script worked is the presence of the newly created text file displayed in Microsoft Notepad as shown in Figure 10-4.

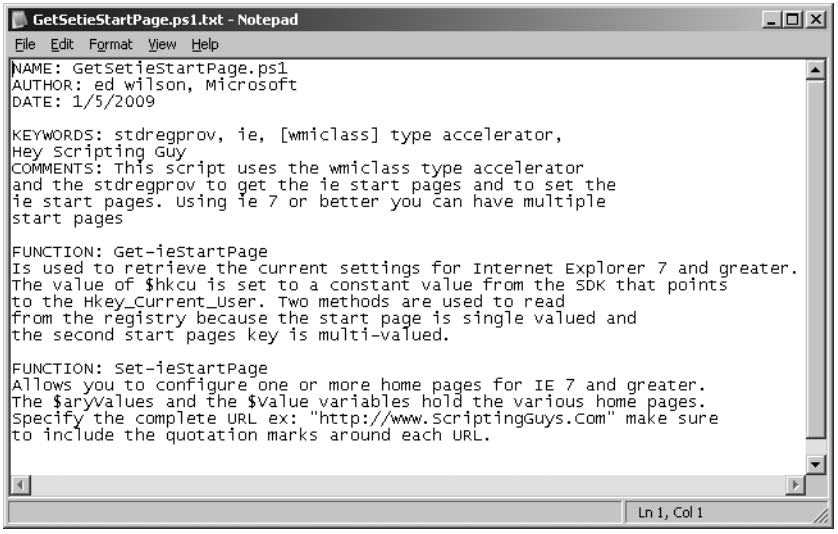


FIGURE 10-4 Comments extracted from a script by the GetCommentsFromScript.ps1 script

The complete GetCommentsFromScript.ps1 script is shown here.

GetCommentsFromScript.ps1

```
Param($Script= $(throw "The path to a script is required."))
Function Get-FileName($Script)
{
 $outputPath = [io.path]::GetTempPath()
 Join-Path -path $outputPath -child (Split-Path $script -leaf)
} #end Get-FileName

Function Remove-outputFile($outputFile)
{
  If(Test-Path -path $outputFile) { Remove-Item $outputFile | Out-Null }
} #end Remove-outputFile

Function Get-Comments($Script,$outputFile)
{
 Get-Content -path $Script |
 Foreach-Object `
  {
    If($_ -match '^\$comment\s?=\s?@"')
     {
      $beginComment = $True
     } #end if match @"
   If($_ -match '"@')
     {
      $beginComment = $False
     } #end if match "@
   If($beginComment -AND $_ -notmatch '@"')
     {
      $_ | Out-File -FilePath $outputFile -append
     } # end if beginComment
  } #end Foreach
} #end Get-Comments

Function Get-outputFile($outputFile)
{
 Notepad $outputFile
} #end Get-outputFile

# *** Entry point to script ***
$outputFile = Get-FileName($script)
Remove-outputFile($outputFile)
Get-Comments -script $script -outputfile $outputFile
Get-outputFile($outputFile)
```

Using Multiple-Line Comment Tags in Windows PowerShell 2.0

Windows PowerShell 2.0 introduces multiple-line comment tags that can be used to comment one or more lines in a script. These comment tags work in a similar fashion to here-strings or HTML tags in that, when you open a comment tag, you must also close the comment tag.

Creating Multiple-Line Comments with Comment Tags

The opening tag is the left angle bracket pound sign (<#), and the closing comment tag is the pound sign right angle bracket (#>). The pattern for the use of the multiline comment is shown here.

```
<# Opening comment tag
First line comment
Additional comment lines
#> Closing comment tag
```

The use of the multiline comment is seen in the Demo-MultilineComment.ps1 script.

Demo-MultilineComment.ps1

```
<#
Get-Command
Get-Help
#>
"The above is a multiline comment"
```

When the Demo-MultilineComment.ps1 script is run, the two cmdlets shown inside the comment tags are not run; the only command that runs is the one outside of the comment block, which prints a string in the console window. The output from the Demo-MultilineComment.ps1 script is shown here.

```
The above is a multiline comment
```

Multiline comment tags do not need to be placed on individual lines. It is perfectly permissible to include the commented text on the line that supplies the comment characters. The pattern for the alternate multiline comment tag placement is shown here.

```
<# Opening comment tag First line comment
Additional comment lines #> Closing comment tag
```

The alternate multiline comment tag placement is shown in MultilineDemo2.ps1.

MultilineDemo2.ps1

```
<# Get-Help
   Get-Command #>
"The above is a multiline comment"
```

NOTE **As a best practice, I prefer to place multiline comment tags on their own individual lines. This format makes the code much easier to read, and it is easier to see where the comment begins and ends.**

Creating Single-Line Comments with Comment Tags

You can use the multiline comment syntax to comment a single line of code, with the advantage being that you do not mix your comment characters. You can use a single comment pattern for all of the comments in the script as shown here.

```
<# Opening comment tag First line comment #> Closing comment tag
```

An example of the single comment pattern in a script is shown in the MultilineDemo3.ps1 script.

MultilineDemo3.ps1

```
<# This is a single comment #>
"The above is a single comment"
```

When using the multiline comment pattern, it is important to keep in mind that anything placed after the end of the closing comment tag is parsed by Windows PowerShell. Only items placed within the multiline comment characters are commented out. However, multiline commenting behavior is completely different from using the pound sign (#) single-line comment character. It is also a foreign concept to users of VBScript who are used to the behavior of the single quote (') comment character in which anything after the character is commented out. A typical-use scenario that generates an error is illustrated in the following example.

```
<# ----------------------------
This example causes an error
#> ----------------------------
```

If you need to highlight your comments in the manner shown in the previous example, you only need to change the position of the last comment tag by moving it to the end of the line to remove the error. The modified comment is shown here.

```
<# --------------------------------
This example does not cause an error
-------------------------------- #>
```

NOTE **No space is required between the pound sign and the following character. I prefer to include the space between the pound sign and the following character as a concession to readability.**

The single pound sign (#) is still accepted for commenting, and there is nothing to prohibit its use. To perform a multiline comment using the single pound sign, you simply place a pound sign in front of each line that requires commenting. This pattern has the advantage of familiarity and consistency of behavior. The fact that it is also backward compatible with Windows PowerShell 1.0 is an added bonus.

```
# First commented line
# additional commented line
# last commented line
```

The 13 Rules for Writing Effective Comments

When adding documentation to a script, it is important that you do not introduce errors. If the comments and code do not match, there is a good chance that both are wrong. Make sure that when you modify the script, you also modify your comments. In this way, both the comments and the script refer to the same information.

Update Documentation When a Script Is Updated

It is easy to forget to update comments that refer to the parameters of a function when you add additional parameters to that function. In a similar fashion, it is easy to ignore the information contained inside the header of the script that refers to dependencies or assumptions within the script. Make sure that you treat both the script and the comments with the same level of attention and importance. In the FindDisabledUserAccounts.ps1 script, the comments in the header seem to apply to the script, but they also seem to miss the fact that the script is using the [ADSISearcher] type accelerator. In fact, the script is a modified script that was used to create a specific instance of the DirectoryServices.DirectorySearcher .NET Framework class and was recently updated. However, the comments were never updated. This oversight might make a user suspicious as to the accuracy of a perfectly useful script. The FindDisabledUserAccounts.ps1 script is shown here.

FindDisabledUserAccounts.ps1

```
# ----------------------------------------------------------------------------
# FindDisabledUserAccounts.ps1
# ed wilson, 3/28/2008
#
# Creates an instance of the DirectoryServices DirectorySearcher .NET
# Framework class to search Active Directory.
# Creates a filter that is LDAP syntax that gets applied to the searcher
```

```
# object. If we only look for class of user, then we also end up with
# computer accounts as they are derived from user class. So we do a
# compound query to also retrieve person.
# We then use the findall method and retrieve all users.
# Next we use the properties property and choose item to retrieve the
# distinguished name of each user, and then we use the distinguished name
# to perform a query and retrieve the UAC attribute, and then we do a
# boolean to compare with the value of 2 which is disabled.
#
# ------------------------------------------------------------------------
#Requires -Version 2.0

$filter = "(&(objectClass=user)(objectCategory=person))"
$users = ([adsiSearcher]$Filter).findall()

 foreach($suser in $users)
  {
   "Testing $($suser.properties.item(""distinguishedname""))"
   $user = [adsi]"LDAP://$($suser.properties.item(""distinguishedname""))"

   $uac=$user.psbase.invokeget("useraccountcontrol")
     if($uac -band 0x2)
       { write-host -foregroundcolor red "`t account is disabled" }
     ELSE
       { write-host -foregroundcolor green "`t account is not disabled" }
  } #foreach
```

Add Comments During the Development Process

When you are writing a script, make sure that you add the comments at the same time you are doing the initial development. Do not wait until you have completed the script to begin writing your comments. When you make comments after writing the script, it is very easy to leave out details because you are now overly familiar with the script and those items that you looked up in documentation now seem obvious. If you add the comments at the same time that you write the script, you can then refer to these comments as you develop the script to ensure that you maintain a consistent approach. This procedure will help with the consistency of your variable names and writing style. The CheckForPdfAndCreateMarker.ps1 script illustrates this consistency problem. In reviewing the code, it seems that the script checks for PDF files, which also seems rather obvious from the name of the script. However, why is the script prompting to delete the files? What is the marker? The only discernable information is that I wrote the script back in December 2008 for a Hey Scripting Guy! article. Luckily, Hey Scripting Guy! articles explain scripts, so at least some documentation actually exists! The CheckForPdfAndCreateMarker.ps1 script is shown here.

CheckForPdfAndCreateMarker.ps1

```
# ----------------------------------------------------------------------------------------
# CheckForPdfAndCreateMarker.ps1
# ed wilson, msft, 12/11/2008
#
# Hey Scripting Guy! 12/29/2008
# ----------------------------------------------------------------------------------------
$path = "c:\fso"
$include = "*.pdf"
$name = "nopdf.txt"
if(!(Get-ChildItem -path $path -include $include -Recurse))
  {
    "No pdf was found in $path. Creating $path\$name marker file."
    New-Item -path $path -name $name -itemtype file -force |
    out-null
  } #end if not Get-Childitem
ELSE
 {
  $response = Read-Host -prompt "PDF files were found. Do you wish to delete <y>
/<n>?"
  if($response -eq "y")
    {
     "PDF files will be deleted."
     Get-ChildItem -path $path -include $include -recurse |
      Remove-Item
    } #end if response
  ELSE
   {
    "PDF files will not be deleted."
   } #end else reponse
 } #end else not Get-Childitem
```

Write for an International Audience

When you write comments for your script, you should attempt to write for an international audience. You should always assume that users who are not overly familiar with the idioms of your native language will be reading your comments. In addition, writing for an international audience makes it easier for automated software to localize the script documentation. Key points to keep in mind when writing for an international audience are to use a simple syntax and to use consistent employee standard terminology. Avoid slang, acronyms, and overly familiar language. If possible, have a colleague who is a non-native speaker review the documentation. In the SearchForWordImages.ps1 script, the comments explain what the script does and also its limitations, such as the fact that it was only tested using Microsoft Office Word 2007. The sentences are plainly written and do not use jargon or idioms. The SearchForWordImages.ps1 script is shown here.

SearchForWordImages.ps1

```
# ------------------------------------------------------------------------
# NAME: SearchForWordImages.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 11/4/2008
#
# KEYWORDS: Word.Application, automation, COM
# Get-Childitem -include, Foreach-Object
#
# COMMENTS: This script searches a folder for doc and
# docx files, opens them with Word and counts the
# number of images embedded in the file.
# It then prints out the name of each file and the
# number of associated images with the file. This script requires
# Word to be installed. It was tested with Word 2007. The folder must
# exist or the script will fail.
#
# ------------------------------------------------------------------------
#The folder must exist and be followed with a trailing \*
$folder = "c:\fso\*"
$include = "*.doc","*.docx"
$word = new-object -comobject word.application
#Makes the Word application invisible. Set to $true to see the application.
$word.visible = $false
Get-ChildItem -path $folder -include $include |
ForEach-Object `
{
 $doc = $word.documents.open($_.fullname)
 $_.name + " has " + $doc.inlineshapes.count + " images in the file"
}
#If you forget to quit Word, you will end up with multiple copies running
#at the same time.
$word.quit()
```

Consistent Header Information

You should include header information at the top of each script. This header information should be displayed in a consistent manner and indeed should be part of your company's scripting standards. Typical information to be displayed is the title of the script, author of the script, date the script was written, version information, and additional comments. Version information does not need to be more extensive than the major and minor versions. This information, as well as comments as to what was added during the revisions, is useful for maintaining a version control for production scripts. An example of adding comments is shown in the WriteBiosInfoToWord.ps1 script.

WriteBiosInfoToWord.ps1

```
# ============================================================================
#
# NAME: WriteBiosInfoToWord.ps1
#
# AUTHOR: ed wilson , Microsoft
# DATE  : 10/30/2008
# EMAIL: Scripter@Microsoft.com
# Version: 1.0
#
# COMMENT: Uses the word.application object to create a new text document
# uses the get-wmiobject cmdlet to query wmi
# uses out-string to remove the "object nature" of the returned information
# uses foreach-object cmdlet to write the data to the word document.
#
# Hey Scripting Guy! 11/11/2008
# ============================================================================

$class = "Win32_Bios"
$path = "C:\fso\bios"

#The wdSaveFormat object must be saved as a reference type.
[ref]$SaveFormat = "microsoft.office.interop.word.WdSaveFormat" -as [type]

$word = New-Object -ComObject word.application
$word.visible = $true
$doc = $word.documents.add()
$selection = $word.selection
$selection.typeText("This is the bios information")
$selection.TypeParagraph()

Get-WmiObject -class $class |
Out-String |
ForEach-Object { $selection.typeText($_) }
$doc.saveas([ref] $path, [ref]$saveFormat::wdFormatDocument)
$word.quit()
```

Document Prerequisites

It is imperative that your comments include information about prerequisites for running the script as well as the implementation of nonstandard programs in the script. For example, if your script requires the use of an external program that is not part of the operating system, you need to include checks within the script to ensure that the program is available when it is called by the script itself. In addition to these checks, you should document the fact that the program is a requirement for running the script. If your script makes assumptions as to the existence of certain directories, you should make a note of this fact. Of course, your script

should use Test-Path to make sure that the directory exists, but you should still document this step as an important precondition for the script. An additional consideration is whether or not you create the required directory. If the script requires an input file, you should add a comment that indicates this requirement as well as add a comment to check for the existence of the file prior to actually calling that file. It is also a good idea to add a comment indicating the format of the input file because one of the most fragile aspects of a script that reads an input file is the actual formatting of that file. The ConvertToFahrenheit_include.ps1 script illustrates adding a note about the requirement of accessing the include file.

ConvertToFahrenheit_include.ps1

```
# -------------------------------------------------------------------------
# NAME: ConvertToFahrenheit_include.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 9/24/2008
# EMAIL: Scripter@Microsoft.com
# Version 2.0
#   12/1/2008 added test-path check for include file
#             modified the way the include file is called
# KEYWORDS: Converts Celsius to Fahrenheit
#
# COMMENTS: This script converts Celsius to Fahrenheit
# It uses command line parameters and an include file.
# If the ConversionFunctions.ps1 script is not available,
# the script will fail.
#
# -------------------------------------------------------------------------
Param($Celsius)
#The $includeFile variable points to the ConversionFunctions.ps1
#script. Make sure you edit the path to this script.
$includeFile = "c:\data\scriptingGuys\ConversionFunctions.ps1"
if(!(test-path -path $includeFile))
  {
   "Unable to find $includeFile"
   Exit
  }
. $includeFile
ConvertToFahrenheit($Celsius)
```

Document Deficiencies

If the script has a deficiency, it is imperative that this is documented. This deficiency may be as simple as the fact that the script is still in progress, but this fact should be highlighted in the comments section of the header to the script. It is quite common for script writers to begin writing a script, become distracted, and then begin writing a new script, all the while forgetting about the original script in progress. When the original script is later found, someone might begin to use the script and be surprised that it does not work as advertised. For

this reason, scripts that are in progress should always be marked accordingly. If you use a keyword, such as *in progress*, then you can write a script that will find all of your work-in-progress scripts. In addition to scripts in progress, you should also highlight any limitations of the script. If a script runs on a local computer but will not run on a remote computer, this fact should be added in the comment section of the header. If a script requires an extremely long time to complete the requested action, this information should be noted. If the script generates errors but completes its task successfully, this information should also be noted so that the user can have confidence in the outcome of the script. A note that indicates why the error is generated also increases the confidence of the user in the original writer. The CmdLineArgumentsTime.ps1 script works but generates errors unless it is used in a certain set of conditions and is called in a specific manner. The comments call out the special conditions, and several INPROGRESS tags indicate the future work required by the script. The CmdLineArgumentsTime.ps1 script is shown here.

CmdLineArgumentsTime.ps1

```
# ============================================================================
#
# NAME: CmdLineArgumentsTime.ps1
# AUTHOR: Ed Wilson , microsoft
# DATE  : 2/19/2009
# EMAIL: Scripter@Microsoft.com
# Version .0
# KEYWORDS: Add-PSSnapin, powergadgets, Get-Date
#
# COMMENT: The $args[0] is unnamed argument that accepts command line input.
# C:\cmdLineArgumentsTime.ps1 23 52
# No commas are used to separate the arguments. Will generate an error if used.
# Requires powergadgets.
# INPROGRESS: Add a help function to script.
# ============================================================================
#INPROGRESS: change unnamed arguments to a more user friendly method
[int]$inthour = $args[0]
[int]$intMinute = $args[1]
#INPROGRESS: find a better way to check for existence  of powergadgets
#This causes errors to be ignored and is used when checking for PowerGadgets
$erroractionpreference = "SilentlyContinue"
#this clears all errors and is used to see if errors are present.
$error.clear()
#This command will generate an error if PowerGadgets are not installed
Get-PSSnapin *powergadgets | Out-Null
#INPROGRESS: Prompt before loading powergadgets
If ($error.count -ne 0)
{Add-PSSnapin powergadgets}

New-TimeSpan -Start (get-date) -end (get-date -Hour $inthour -Minute $intMinute) |
Out-Gauge -Value minutes -Floating -refresh 0:0:30  -mainscale_max 60
```

Avoid Useless Information

Inside the code of the script itself, you should avoid comments that provide useless or irrelevant information. Keep in mind that you are writing a script and providing documentation for the script and that such a task calls for technical writing skills, not creative writing skills. While you might be enthralled with your code in general, the user of the script is not interested in how difficult it was to write the script. However, it is useful to explain why you used certain constructions instead of other forms of code writing. This information, along with the explanation, can be useful to people who might modify the script in the future. You should therefore add internal comments only if they will help others to understand how the script actually works. If a comment does not add value, the comment should be omitted. The DemoConsoleBeep.ps1 script contains numerous comments in the body of the script. However, several of them are obvious, and others actually duplicate information from the comments section of the header. There is nothing wrong with writing too many comments, but it can be a bit excessive when a one-line script contains 20 lines of comments, particularly when the script is very simple. The DemoConsoleBeep.ps1 script is shown here.

DemoConsoleBeep.ps1

```
# -----------------------------------------------------------------------
# NAME: DemoConsoleBeep.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 4/1/2009
#
# KEYWORDS: Beep
#
# COMMENTS: This script demonstrates using the console
# beep. The first parameter is the frequency between
# 37..32767. above 7500 is barely audible. 37 is the lowest
# note it will play.
# The second parameter is the length of time
#
# -----------------------------------------------------------------------
#this construction creates an array of numbers from 37 to 3200
#the % sign is an alias for Foreach-Object
#the $_ is an automatic variable that refers to the current item
#on the pipeline.
#the semicolon causes a new logical line
#the double colon is used to refer to a static method
#the $_ in the method is the number on the pipeline
#the second number is the length of time to play the beep
37..32000 | % { $_ ; [console]::beep($_ , 1) }
```

Document the Reason for the Code

While it is true that good code is readable and that a good developer is able to understand what a script does, some developers might not understand why a script is written in a certain manner or why a script works in a particular fashion. In the DemoConsoleBeep2.ps1 script, extraneous comments have been removed. Essential information about the range that the console beep will accept is included, but the redundant information is deleted. In addition, a version history is added because significant modification to the script was made. The DemoConsoleBeep2.ps1 script is shown here.

DemoConsoleBeep2.ps1

```
# ------------------------------------------------------------------------
# NAME: DemoConsoleBeep2.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 4/1/2009
# VERSION 2.0
# 4/4/2009 cleaned up comments. Removed use of % alias. Reformatted.
#
# KEYWORDS: Beep
#
# COMMENTS: This script demonstrates using the console
# beep. The first parameter is the frequency. Allowable range is between
# 37..32767. A number above 7500 is barely audible. 37 is the lowest
# note the console beep will play.
# The second parameter is the length of time.
#
# ------------------------------------------------------------------------

37..32000 |
Foreach-Object { $_ ; [console]::beep($_ , 1) }
```

Use of One-Line Comments

You should use one-line comments that appear prior to the code that is being commented to explain the specific purpose of variables or constants. You should also use one-line comments to document fixes or workarounds in the code as well as to point to the reference information explaining these fixes or workarounds. Of course, you should strive to write code that is clear enough to not require internal comments. Do not add comments that simply repeat what the code already states. Add comments to illuminate the code but not to elucidate the code. The GetServicesInSvchost.ps1 script uses comments to discuss the logic of mapping the *handle* property from the *Win32_Process* class to the *ProcessID* property from the *Win32_Service* WMI class to reveal which services are using which instance of the Svchost process. The GetServicesInSvchost.ps1 script is shown here.

GetServicesInSvchost.ps1

```
# ------------------------------------------------------------------------
# NAME: GetServicesInSvchost.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 8/21/2008
#
# KEYWORDS: Get-WmiObject, Format-Table,
# Foreach-Object
#
# COMMENTS: This script creates an array of WMI process
# objects and retrieves the handle of each process object.
# According to MSDN the handle is a process identifier. It
# is also the key of the Win32_Process class. The script
# then uses the handle which is the same as the processID
# property from the Win32_service class to retrieve the
# matches.
#
# HSG 8/28/2008
# ------------------------------------------------------------------------

$aryPid = @(Get-WmiObject win32_process -Filter "name='svchost.exe'") |
  Foreach-Object { $_.Handle }

"There are " + $arypid.length + " instances of svchost.exe running"

foreach ($i in $aryPID)
{
 Write-Host "Services running in ProcessID: $i" ;
 Get-WmiObject win32_service -Filter " processID = $i" |
 Format-Table name, state, startMode
}
```

Avoid End-of-Line Comments

You should avoid using end-of-line comments. The addition of such comments to your code has a severely distracting aspect to structured logic blocks and can cause your code to be more difficult to read and maintain. Some developers try to improve on this situation by aligning all of the comments at a particular point within the script. While this initially looks nice, it creates a maintenance nightmare because each time the code is modified, you run into the potential for a line to run long and push past the alignment point of the comments. When this occurs, it forces you to move everything over to the new position. Once you do this a few times, you will probably realize the futility of this approach to commenting internal code. One additional danger of using end-of-line comments when working with Windows PowerShell is that, due to the pipelining nature of language, a single command might stretch

out over several lines. Each line that ends with a pipeline character continues the command to the next line. A comment character placed after a pipeline character will break the code as shown here, where the comment is located in the middle of a logical line of code. This code will not work.

```
Get-Process | #This cmdlet obtains a listing of all processes on the computer
Select-Object –property name
```

A similar situation also arises when using the named parameters of the `ForEach-Object` cmdlet as shown in the SearchAllComputersInDomain.ps1 script. The backtick (`) character is used for line continuation, which allows placement of the *–Begin*, *–Process*, and *–End* parameters on individual lines. This placement makes the script easier to read and understand. If an end-of-line comment is placed after any of the backtick characters, the script will fail. The SearchAllComputersInDomain.ps1 script is shown here.

SearchAllComputersInDomain.ps1

```
$Filter = "ObjectCategory=computer"
$Searcher = New-Object System.DirectoryServices.DirectorySearcher($Filter)
$Searcher.Findall() |
Foreach-Object `
  -Begin { "Results of $Filter query: " } `
  -Process { $_.properties ; "`r"} `
  -End { [string]$Searcher.FindAll().Count + " $Filter results were found" }
```

Document Nested Structures

The previous discussion about end-of-line comments should not be interpreted as dismissing comments that document the placement of closing curly brackets. In general, you should avoid creating deeply nested structures, but sometimes they cannot be avoided. The use of end-of-line comments with closing curly brackets can greatly improve the readability and maintainability of your script. As shown in the Get-MicrosoftUpdates.ps1 script, the closing curly brackets are all tagged.

Get-MicrosoftUpdates.ps1

```
# ------------------------------------------------------------------------
# NAME: Get-MicrosoftUpdates.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 2/25/2009
#
# KEYWORDS: Microsoft.Update.Session, com
#
# COMMENTS: This script lists the Microsoft Updates
# you can select a certain number, or you can choose
# all of the updates.
#
# HSG 3-9-2009
```

```
# ------------------------------------------------------------------------
Function Get-MicrosoftUpdates
{
  Param(
        $NumberOfUpdates,
        [switch]$all
       )
  $Session = New-Object -ComObject Microsoft.Update.Session
  $Searcher = $Session.CreateUpdateSearcher()
  if($all)
    {
      $HistoryCount = $Searcher.GetTotalHistoryCount()
      $Searcher.QueryHistory(1,$HistoryCount)
    } #end if all
  Else
    {
      $Searcher.QueryHistory(1,$NumberOfUpdates)
    } #end else
} #end Get-MicrosoftUpdates

# *** entry point to script ***

# lists the latest update
# Get-MicrosoftUpdates -NumberofUpdates 1

# lists All updates
Get-MicrosoftUpdates -all
```

Use a Standard Set of Keywords

When adding comments that indicate bugs, defects, or work items, you should use a set of keywords that is consistent across all scripts. This would be a good item to add to your corporate scripting guidelines. In this way, a script can easily be developed that will search your code for such work items. If you maintain source control, then a comment can be added when these work items are fixed. Of course, you would also increment the version of the script with a comment relating to the fix. In the CheckEventLog.ps1 script, the script accepts two command-line parameters. One parameter is for the event log to query, and the other is for the number of events to return. If the user selects the security log and is not running the script as an administrator, an error is generated that is noted in the comment block. Because this scenario could be a problem, the outline of a function to check for admin rights has been added to the script as well as code to check for the log name. A number of TODO: tags are added to the script to mark the work items. The CheckEventLog.ps1 script is shown here.

CheckEventLog.ps1

```
# ------------------------------------------------------------------------
# NAME: CheckEventLog.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 4/4/2009
#
# KEYWORDS: Get-EventLog, Param, Function
#
# COMMENTS: This accepts two parameters the logname
# and the number of events to retrieve. If no number for
# -max is supplied it retrieves the most recent entry.
# The script fails if the security log is targeted and it is
# not run with admin rights.
# TODO: Add function to check for admin rights if
# the security log is targeted.
# ------------------------------------------------------------------------
Param($log,$max)
Function Get-log($log,$max)
{
 Get-EventLog -logname $log -newest $max
} #end Get-Log

#TODO: finish Get-AdminRights function
Function Get-AdminRights
{
#TODO: add code to check for administrative
#TODO: rights. If not running as an admin
#TODO: if possible add code to obtain those rights
} #end Get-AdminRights

If(-not $log) { "You must specify a log name" ; exit}
if(-not $max) { $max = 1 }
#TODO: turn on the if security log check
# If($log -eq "Security") { Get-AdminRights ; exit }
Get-Log -log $log -max $max
```

Document the Strange and Bizarre

The last item that should be commented in your documentation is anything that looks strange. If you use a new type of construction that you have not used previously in other scripts, you should add a comment to the effect. A good comment should also indicate the previous coding construction as an explanation. In general, it is not a best practice to use code that looks strange simply to show your dexterity or because it is an elegant solution; rather, you should strive for readable code. However, when you discover a new construction

that is cleaner and easier to read, albeit a somewhat novel approach, you should always add a comment to highlight this fact. If the new construction is sufficiently useful, then it should be incorporated into your corporate scripting guidelines as a design pattern. In the GetProcessesDisplayTempFile.ps1 script, a few unexpected items crop up. The first is the *GetTempFileName* static method from the *Io.Path* .NET Framework class. Despite the method's name, *GetTempFileName* both creates a temporary file name as well as a temporary file itself. The second technique is much more unusual. When the temporary file is displayed via Notepad, the result of the operation is pipelined to the `Out-Null` cmdlet. This operation effectively halts the execution of the script until the Notepad application is closed. This "trick" does not conform to expected behavior, but it is a useful design pattern for those wanting to remove temporary files once they have been displayed. As a result, both features of the GetProcessDisplayTempFile.ps1 script are documented as shown here.

GetProcessesDisplayTempFile.ps1

```
# ----------------------------------------------------------------------
# NAME: GetProcessesDisplayTempFile.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 4/4/2009
# VERSION 1.0
#
# KEYWORDS: [io.path], GetTempFileName, out-null
#
# COMMENTS: This script creates a temporary file,
# obtains a collection of process information and writes
# that to the temporary file. It then displays that file via
# Notepad and then removes the temporary file when
# done.
#
# ----------------------------------------------------------------------
#This both creates the file name as well as the file itself
$tempFile = [io.path]::GetTempFileName()
Get-Process >> $tempFile
#Piping the Notepad filename to the Out-Null cmdlet halts
#the script execution
Notepad $tempFile | Out-Null
#Once the file is closed the temporary file is closed and it is
#removed
Remove-Item $tempFile
```

INSIDE TRACK

Teaching Your Scripts to Communicate

Peter Costantini, Microsoft Scripting Guy Emeritus

If code was read only by computers, we could only write 1s and 0s. Even though developers would quickly go blind and insane, there's a new class of computer science majors graduating every year. Of course, the reality is that code must also be read by humans, and programming languages have been developed to mediate between humans and machines.

If one developer could write, debug, test, maintain, and field support calls for all of the code for an application, then it wouldn't be very important whether the programming language was easy for others to understand. A brilliant loner could decide to write in an obscure dialect of Lisp and name the variables and procedures in Esperanto, and that would be fine as long as the code worked.

However, that programing language may not be so fine five years later. By then, the developer's Lisp and Esperanto are a little rusty. Suddenly a call comes in that the now mission-critical application is crashing inexplicably and losing the firm billions of dollars.

"What's a few billion dollars these days? Maybe I'll get a bonus," I hear you muttering under your breath. Anyway, you're not a developer: you're a system engineer who's trying to use scripts to automate some of your routine tasks and to troubleshoot. You thought the whole point of scripting was to let you write quick and dirty code to get a task done in a hurry.

Yes, that is a big benefit of scripting. When you first write a script to solve a problem, you're probably not concerned about producing beautiful-looking, or even comprehensible, code. You just want to make sure that it runs as expected and makes the pain stop.

However, once you decide that the script is a keeper and that you're going to run it as a scheduled task at three every Monday morning, the equation starts to change. At this point, like it or not, you really are a developer. Windows PowerShell is a programming language, albeit a dynamic one, and any code that plays an ongoing role in the functioning of your organization needs to be treated as something more than chewing gum and baling wire.

Furthermore, regardless of your personal relationship with your scripts, you probably work as part of a team, right? Other people on your team might write scripts, too. In any case, these people most likely have to run your scripts and figure out what they do. You can see where I'm going with this. But if it produces a blinding flash of insight, that's all the better for your career and your organization.

The goal is to make your scripts transparent. Your code—and the environment in which it runs—should communicate to your teammates everything they need to know to understand what your script is doing, how to use it successfully, and how to troubleshoot it if problems arise (Murphy's Law has many scripting corollaries). Clarity and readability are virtues; terseness and ambiguity are not. Consistent, descriptive variable names and white space do not make the code run any slower, but they can make the script more readable. Begin to look at transparency as an insurance policy against receiving a frantic call on your cell phone when you're lying on a beach in Puerto Vallarta sipping a margarita.

Yet, this is not just a technical and social imperative: it's an economic one as well. IT departments are pushing hard to become strategic assets rather than cost centers. The sprawling skeins of code, scripts, and all that run their operations can earn or lose figures followed by many zeros and make the difference between budget increases and layoffs. Okay, at least this year, adding good documentation to your scripts can make the budget cuts smaller.

Additional Resources

- The TechNet Script Center at *http://www.microsoft.com/technet/scriptcenter* contains numerous examples of Windows PowerShell scripts, as well as some sample documentation templates.
- Take a look at the *Windows PowerShell™ Scripting Guide* (Microsoft Press, 2008).
- The Tweakomatic can be downloaded from the TechNet Script Center at *http://www.microsoft.com/technet/scriptcenter*.
- Refer to "How Can I Delete and Manage PDF Files?" at *http://www.microsoft.com /technet/scriptcenter/resources/qanda/dec08/hey1229.mspx*.
- Refer to "How Can I Create a Microsoft Word Document from WMI Information?" at *http://www.microsoft.com/technet/scriptcenter/resources/qanda/nov08/hey1111.mspx*.
- Refer to *Windows Scripting with WMI: Self-Paced Learning Guide* (Microsoft Press, 2006).
- Script documentation guidelines are discussed in the *Microsoft Windows 2000 Scripting Guide* (Microsoft Press, 2003). This book is available online at *http://www.microsoft.com /technet/scriptcenter/guide/sas_sbp_mybu.mspx*.
- On the companion media, you will find all of the scripts referred to in this chapter.

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

About the Author

Ed Wilson is one of the Microsoft Scripting Guys and a well-known scripting expert. He writes the daily Hey Scripting Guy! article for the Scripting Guys blog on TechNet, as well as a weekly blog posting for Microsoft Press. He has also spoken at the TechEd technical conference and at Microsoft internal TechReady conferences. Ed is a Microsoft Certified Trainer who has delivered a popular Windows PowerShell workshop to Microsoft Premier Customers worldwide. He has written eight books, including five on Windows scripting, and all have been published by Microsoft Press. He has also contributed to nearly one dozen other books.

Ed holds more than twenty industry certifications, including Microsoft Certified Systems Engineer (MCSE) and Certified Information Systems Security Professional (CISSP). Prior to working for Microsoft, he was a senior consultant for a Microsoft Gold Certified Partner and specialized in Active Directory design and Microsoft Exchange Server implementation. In his spare time, Ed enjoys woodworking, underwater photography, and scuba diving.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

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

Thank you in advance for your input!

Microsoft® Press

Stay in touch!

To subscribe to the *Microsoft Press® Book Connection Newsletter*—for news on upcoming books, events, and special offers—please visit:

microsoft.com/learning/books/newsletter