PowerShell 101

The No-Nonsense Beginner's Guide to PowerShell

Mike F Robbins

PowerShell 101

The No-Nonsense Beginner’s Guide to PowerShell

Mike F Robbins

This book is for sale at http://leanpub.com/powershell101

This version was published on 2018-09-04

Contents

Preface

In 2007, I was working as a senior level engineer providing technical support for three companies who were all early adopters of Exchange Server 2007. A number of configuration settings and administrative tasks that needed to be performed on their Exchange Servers couldn't be accomplished in the GUI (Graphical User Interface), so I inadvertently became a dangerous script copy and paster. Years later, I told Ed Wilson, the Scripting Guy, that I had bought his PowerShell 1.0 Step by Step book and it's the book I should have read but never had time or so I thought.

When you say you don't have time for something, what you really mean is that it's not something important enough to you to make time for it. This reminds me of the funny pictures I've seen tweeted out where Neanderthals are pushing something around with square wheels on it and are too busy to make time to learn about round wheels. Being too busy pointing and clicking in the GUI instead of taking time to learn PowerShell is just as inefficient and instead of being funny it's just sad. Are you too busy?

When PowerShell 2.0 released, I could really see that Microsoft was going somewhere with PowerShell so I decided to make time to learn it. I know, you're too busy at work to have time to learn PowerShell. Guess what, we all are. I was only able to spend a small amount of time at work learning PowerShell and I would have never learned it well at the rate I was going.

You're in control of your own destiny. I decided to dedicate an enormous amount of my personal time to learning PowerShell, often times staying up to the wee hours of the night while still having to be at work early the next morning. I know, you have a family. I'm a big believer in work-life balance. I'm married with three children and never miss out on family activities if at all possible. Somehow, most people who are too busy due to family responsibilities always seem to know what happened on the most recent television shows. I know, there's no budget for training. While some employers do offer paid training for their staff, ultimately it's your responsibility to keep your skills up to date. There are plenty of free resources available in the form of books, videos, podcasts, and blog articles. There are even free in-person technology events such as user group meetings and PowerShell Saturdays where you can not only learn, but network with others in the industry. See Appendix B of this book for a list of resources. Consider multitasking. Read a PowerShell book or watch a training video while on the treadmill at the gym. Listen to a podcast during your commute.

I not only learned PowerShell 2.0, I learned it well. Anything worth learning is worth learning well. I competed in the beginner category of the Scripting Games in 2012 (the last year that Ed Wilson hosted it). I led the competition during most of the events and ended up finishing in third place overall.

Some of the improvements in PowerShell version 3.0 made it much easier to use. I competed in the advanced category in the 2013 Scripting Games (the first year that PowerShell.org hosted them) and I won.

Once you figure something out, help others by sharing your knowledge. In 2014, I was awarded Microsoft's prestigious MVP award on Windows PowerShell for my community activities (blogging, speaking, user group, etc.), and have been re-awarded every year since.

From a syntax standpoint, it's very easy to move between systems running PowerShell version 3.0, 4.0, 5.0, and 5.1 without having to relearn anything with the exception of the new functionality included in the newer versions (the syntax is the same).

You'll often see the micro symbol used as my electronic signature. The backstory is that I once worked for a company who used first name and last initial for user names. When I started with the company Miker was already taken so they assigned me the user name Mikero which I pronounced "micro" and eventually adopted the micro symbol as my electronic signature.

µ

Disclaimer

All code examples shown in this book have been tested by the author and every effort has been made to ensure that they are error free, but since every environment is different, they should not be run in a production environment without thoroughly testing them first. It is recommended that you use a lab environment for working through the code examples shown throughout this book.

The disclaimer below is provided simply because someone somewhere will not follow this recommendation and in the event that they do experience problems or a RGE (resume generating event), they have no one to blame but themselves.

Introduction

What is PowerShell?

PowerShell, at least as of version 5.1, is an easy to use command-line shell and scripting environment for automating administrative tasks of Windows based systems.

Who is this book for?

This is an entry-level book for anyone wanting to learn PowerShell.

This book focuses on PowerShell version 5.1 in a Microsoft Active Directory domain environment running on Windows 10 Anniversary Edition (build 1607) and Windows Server 2016. All currently supported versions of Windows and previous versions of PowerShell beginning with PowerShell version 1.0 are briefly mentioned where applicable.

PowerShell version 6.0 is currently in alpha (pre-beta). It can be run on Linux and macOS in addition to Windows. It's not covered in this book since it hasn't been released for use in a production environment.

About this book

Before PowerShell, I spent the first third of my career as an IT Pro pointing and clicking in the GUI. I decided to write this book to save IT Pros from themselves who are reluctant to learn PowerShell and still pointing and clicking in the GUI.

This book is a collection of what I wish someone would have told me when I started learning PowerShell, along with the tips, tricks, and best practices that I've learned while using PowerShell during the past 10 years.

Instead of providing an enormous amount of information, this book attempts to provide a balance of enough information to be successful for someone who is just getting started with PowerShell without overwhelming them with information overload. Each chapter contains a section that references specific help topics in PowerShell for those who want to know more about the information covered in that particular chapter.

"Only a fool learns from his own mistakes. The wise man learns from the mistakes of others." - Otto von Bismarck

About the cover image

The cover image is a photo of a daylily that my sister took years ago. I would like to thank her for allowing me to use it on the cover of this book.

What does a flower have to do with PowerShell? Flowers grow from the ground up. My goal with this book is to help you build your PowerShell skills from the ground up and be well on your way to writing PowerShell code that's as beautiful as the flower on the cover of this book by the time you finish it.

About the author

Mike F Robbins is a Microsoft MVP on Windows PowerShell and a SAPIEN Technologies MVP. He is a co-author of Windows PowerShell TFM 4th Edition and is a contributing author of a chapter in the PowerShell Deep Dives book. Mike has written guest blog articles for the Hey, Scripting Guy! Blog, PowerShell Magazine, and PowerShell.org. He is the winner of the advanced category in the 2013 PowerShell Scripting Games. Mike is also the leader and co-founder of the Mississippi PowerShell User Group. He blogs at mikefrobbins.com and can be found on twitter @mikefrobbins.

About the technical editor

Tommy Maynard is a Senior Systems Administrator with a passion for PowerShell. He has over 15 years of experience in Information Technology, and finally feels as though he's found his calling. Luckily for him, PowerShell works right alongside the technologies he has long supported. His goal is to help educate and inspire people that work in his industry to embrace PowerShell, and in general, scripting and automation. Tommy lives in Tucson, Arizona with his wife and two kids. He blogs at tommymaynard.com and can be found on Twitter @thetommymaynard.

Conventions used in this book

The chapters in this book are designed using the Pomodoro technique. They are written so they can be read in twenty-five minutes. Be sure to take a five minute break after each chapter and a longer break after four chapters if you decide to read that many at once.

Code examples in this book have line numbers and the output of the code immediately follows the code example but does not have line numbers.

Lab environment

Windows 10 Anniversary Edition (build 1607) and Windows Server 2016 both of which have PowerShell version 5.1 installed by default are used throughout this book. If you’re using a different version of Windows and/or PowerShell, your results may differ from those shown in this book.

Chapter 1 - Getting Started with PowerShell

I often find that presenters at conferences and user group meetings already have PowerShell running when they start entry-level presentations. This book begins by answering the questions I've heard attendees who haven't previously used PowerShell ask in those sessions.

Specifically, this chapter will focus on where to find and how to launch PowerShell along with solving a couple of the initial pain points that new users often experience with PowerShell. Be sure to follow along and walkthrough the examples shown in this chapter on your Windows 10 lab environment computer.

What do I need to get started with PowerShell?

All modern versions of Windows operating systems ship with PowerShell installed. Table 1-1 shows the default version and supported versions of PowerShell for each Windows operating system that Microsoft currently supports.

Windows Operating System Version	Default PowerShell Version	Supported PowerShell Versions
Windows Vista	1.0*	1.0, 2.0
Server 2008	1.0*	1.0, 2.0, 3.0
Windows 7	2.0	2.0, 3.0, 4.0, 5.0, 5.1
Server 2008 R2	2.0	2.0, 3.0, 4.0, 5.0, 5.1
Windows 8	3.0	3.0
Server 2012	3.0	3.0, 4.0, 5.0, 5.1
Windows 8.1	4.0	4.0, 5.0, 5.1
Server 2012 R2	4.0	4.0, 5.0, 5.1
Windows 10	5.0	5.0
Windows 10 (Anniversary Update)	5.1	5.1
Server 2016	5.1	5.1

Table 1-1

The bits for PowerShell version 1.0 exist on Windows Vista and Server 2008 (non-R2) but the PowerShell feature is not enabled on those operating systems. PowerShell is not supported on the server core installation (no-GUI) version of Server 2008 (non-R2). PowerShell is also not enabled by default on the server core installation of Server 2008 R2.

Where do I find PowerShell?

The easiest way to find PowerShell on Windows 10 is to simply type PowerShell into the search bar as shown in Figure 1-1.

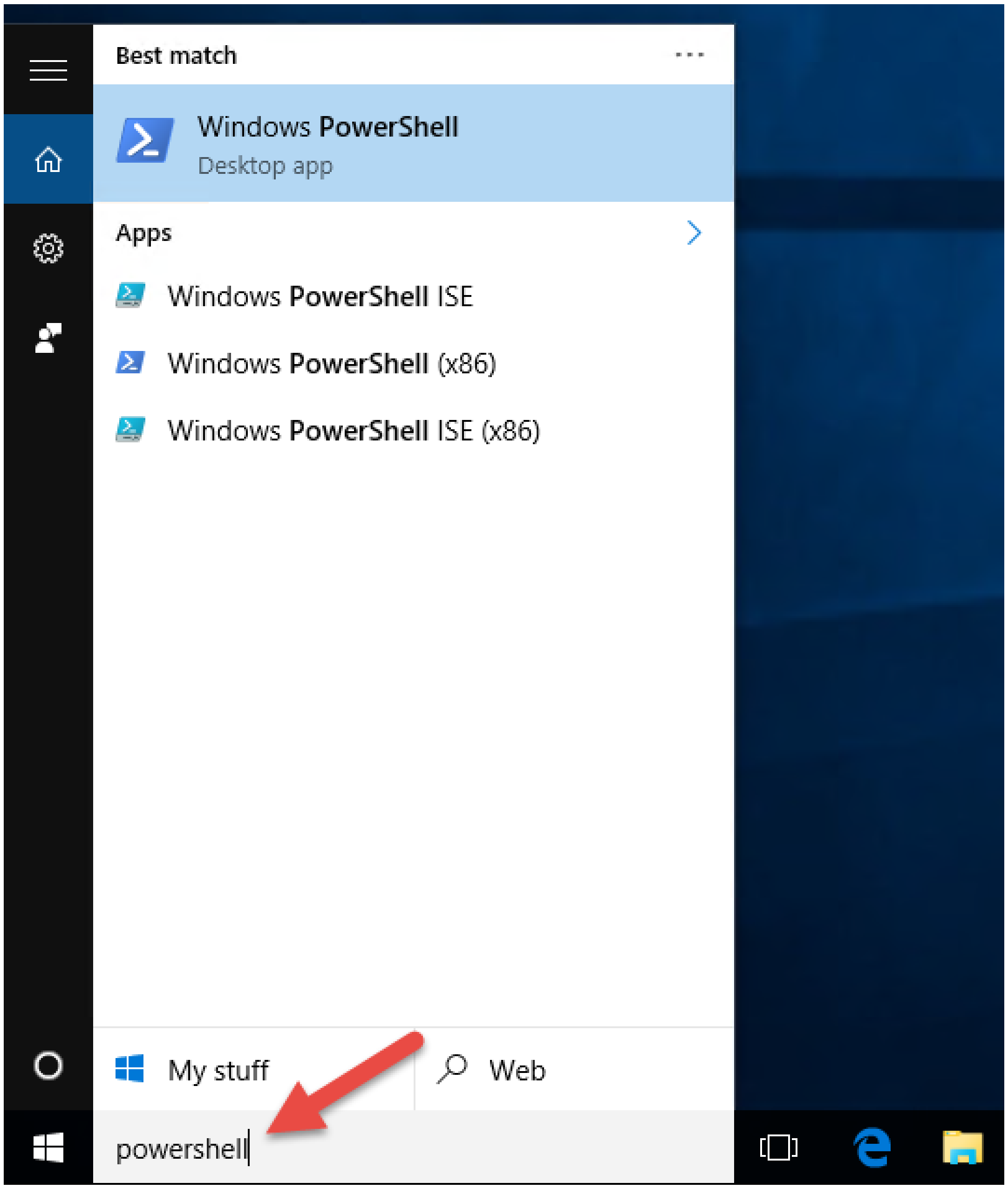


Figure 1-1

Notice that four different shortcuts for PowerShell are shown in Figure 1-1. The computer used for demonstration purposes in this book is running the 64-bit version of Windows 10 so there's a 64-bit version of the PowerShell console and the PowerShell ISE (Integrated Scripting Environment), and a

32-bit version of each one as denoted by the (x86) suffix on the shortcuts. If you happen to be running a 32-bit version of Windows 10, you'll only have two shortcuts and although those won't have the (x86) suffix, they are 32-bit versions. If you have a 64-bit operating system, my recommendation is to run the 64-bit version of PowerShell unless you have a specific reason for running the 32-bit version.

PowerShell can be accessed from the Start screen on Windows 8, 8.1, Server 2012, and Server 2012 R2. Simply start typing in PowerShell on the start screen and Windows will automatically open the search menu as shown in Figure 1-2.

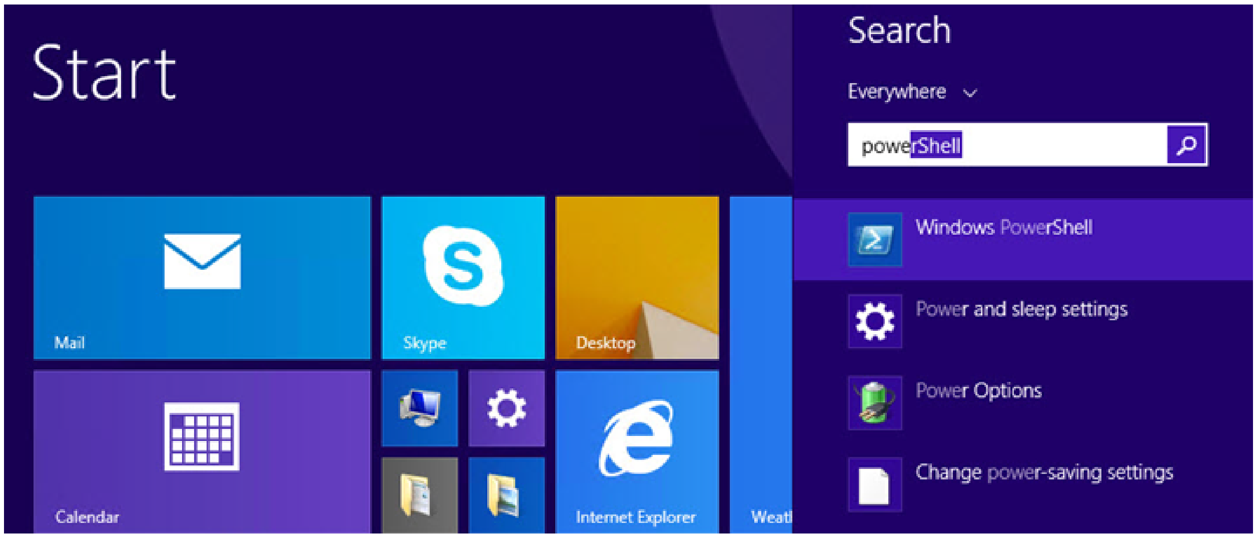


Figure 1-2

It can be accessed from the start menu on Windows 7, Server 2008 R2, and prior operating systems. Navigate to "Start > All Programs > Accessories > Windows PowerShell" as shown in Figure 1-3.

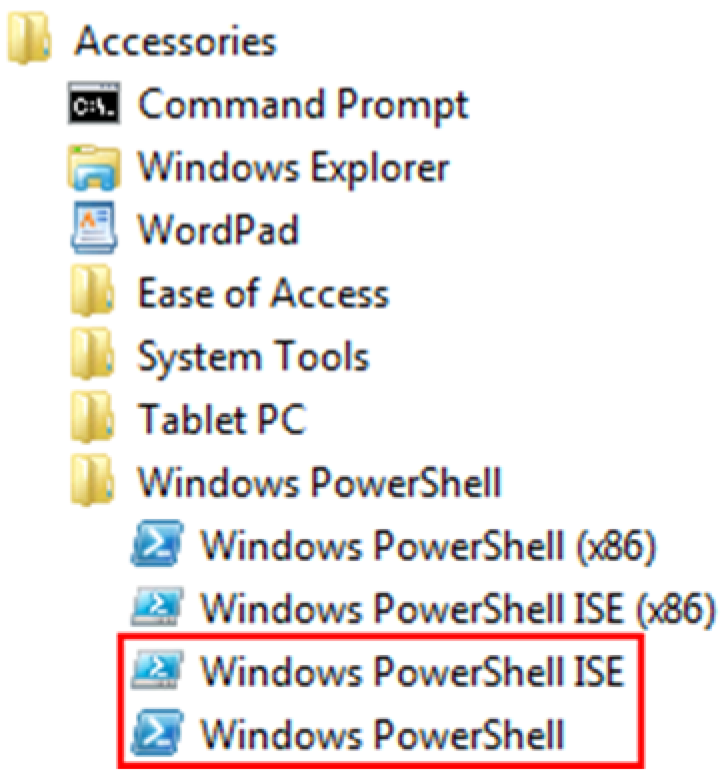
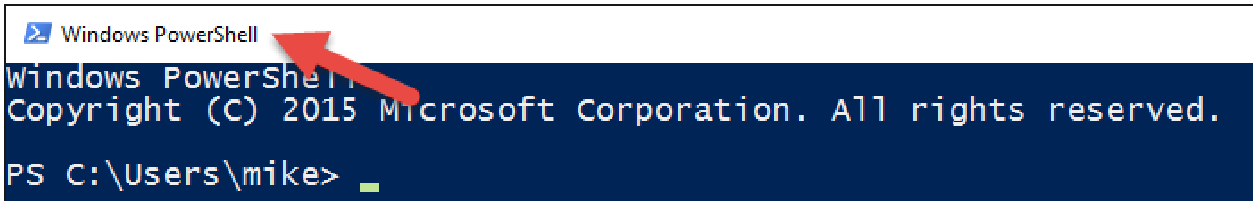


Figure 1-3

How do I launch PowerShell?

In the production enterprise environments that I support, I use three different Active Directory user accounts and I've mirrored those accounts in the lab environment used in this book. I log into the Windows 10 computer as a domain user who is not a domain or local administrator.

I've launched the PowerShell console by simply clicking on the "Windows PowerShell" shortcut as shown in Figure 1-1.



```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

PS C:\Users\mike> _
```

Figure 1-4

Notice that the title bar of the PowerShell console says "Windows PowerShell" as shown in Figure 1-4. Some commands will run fine, but the problem this causes is PowerShell is unable to participate in UAC (User Access Control). That means it's unable to prompt for elevation when performing tasks on the local system that require the approval of an administrator and a cryptic error message will be generated:

```
PS C:\> Get-Service -Name W32Time | Stop-Service
```

```
Stop-Service : Service 'Windows Time (W32Time)' cannot be stopped due to the following
error: Cannot open W32Time service on computer '.'.
At line:1 char:29
+ Get-Service -Name W32Time | Stop-Service
+
    + CategoryInfo          : CloseError: (System.ServiceProcess.ServiceController:Servi
   ceController) [Stop-Service], ServiceCommandException
    + FullyQualifiedErrorId : CouldNotStopService,Microsoft.PowerShell.Commands.StopServ
   iceCommand

PS C:\>
```

The solution to this problem is to run PowerShell as a domain user who is a local administrator. This is how my second domain user account is configured. Using the principal of least privilege, this account should NOT be a domain administrator, or have any elevated privileges in the domain.

Close PowerShell. Relaunch the PowerShell console, except this time right-click on the “Windows PowerShell” shortcut and select “Run as administrator” as shown in Figure 1-5.

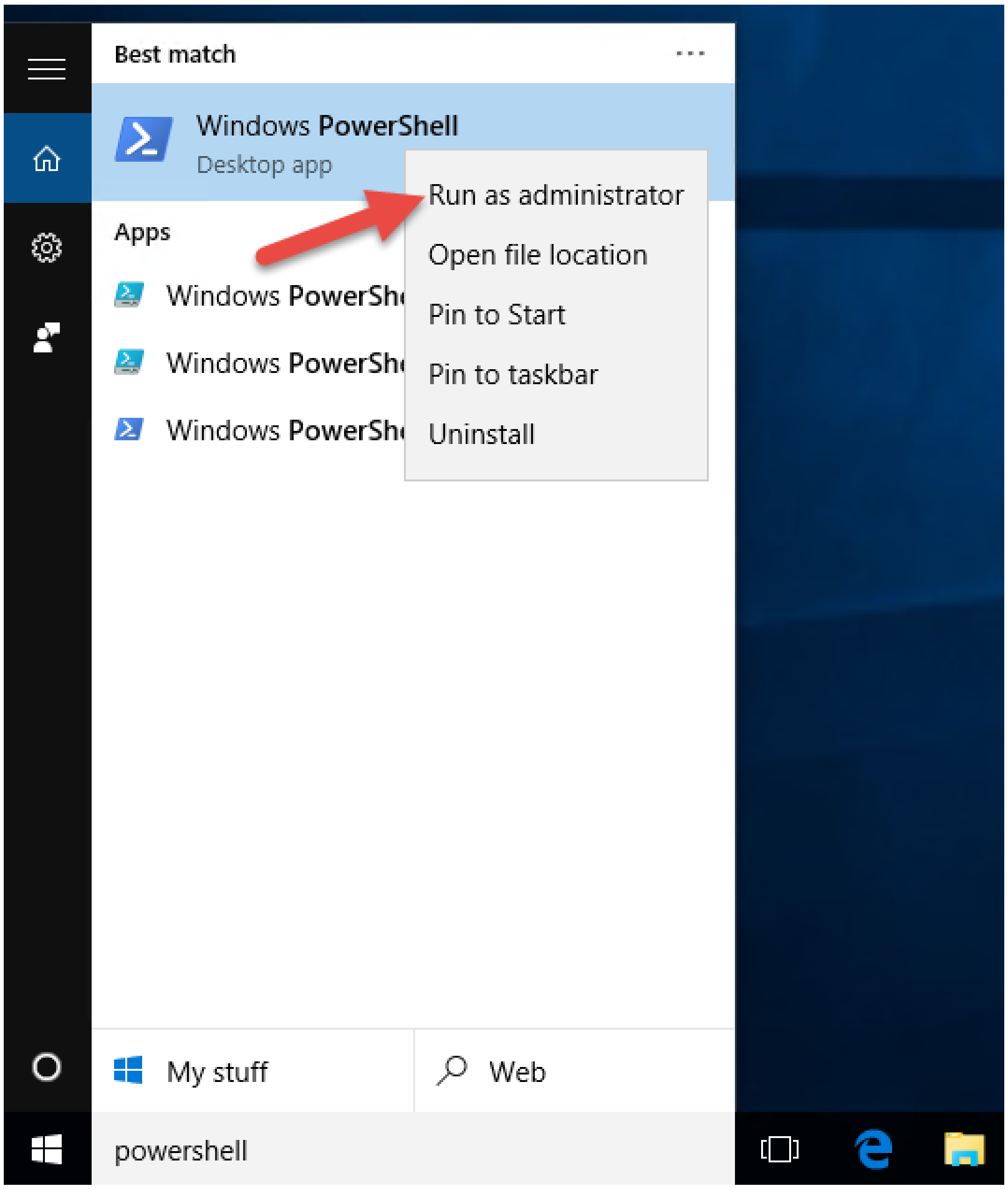
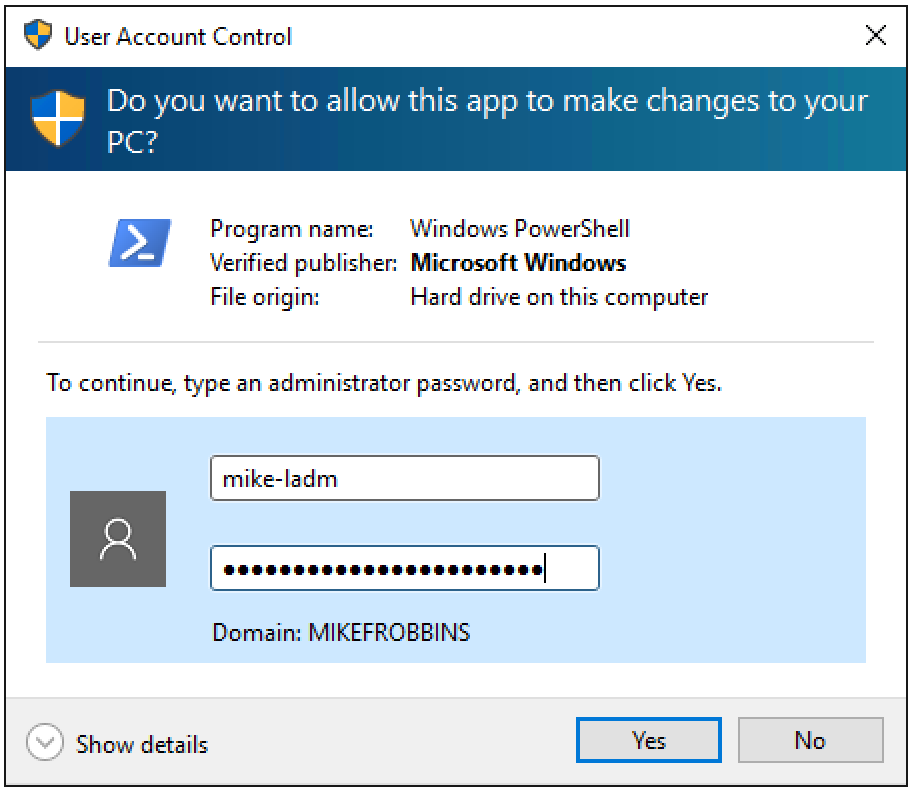


Figure 1-5

If you're logged into Windows as a normal user, you'll be prompted for credentials. I'll enter the credentials for my user account who is a domain user and local admin as shown in Figure 1-6.



Do you want to allow this app to make changes to your PC?

Program name: Windows PowerShell
Verified publisher: **Microsoft Windows**
File origin: Hard drive on this computer

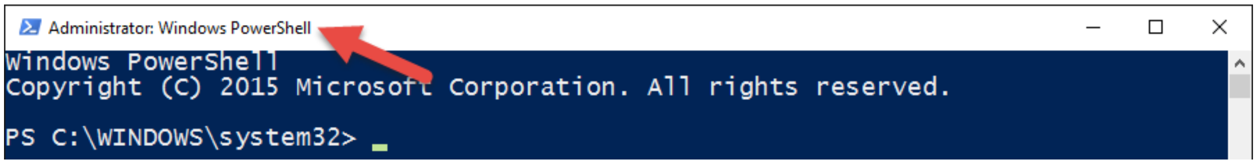
To continue, type an administrator password, and then click Yes.

mike-ladm

Domain: MIKEFROBBINS

Figure 1-6

Once PowerShell is relaunched as an administrator, the title bar should say “Administrator: Windows PowerShell” as shown in Figure 1-7.



```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

PS C:\WINDOWS\system32> _
```

Figure 1-7

Now that PowerShell is being run elevated as a local administrator, UAC will no longer be a problem when a command is run on the local computer that would normally require a prompt for elevation. Keep in mind though that any command run from this elevated instance of the PowerShell console, also runs elevated.

To simplify finding PowerShell and launching it as an administrator, I recommend pinning it to the taskbar and setting it to automatically launch as an admin each time it’s run.

Search for PowerShell again, except this time right-click on it and select “Pin to taskbar” as shown in Figure 1-8.

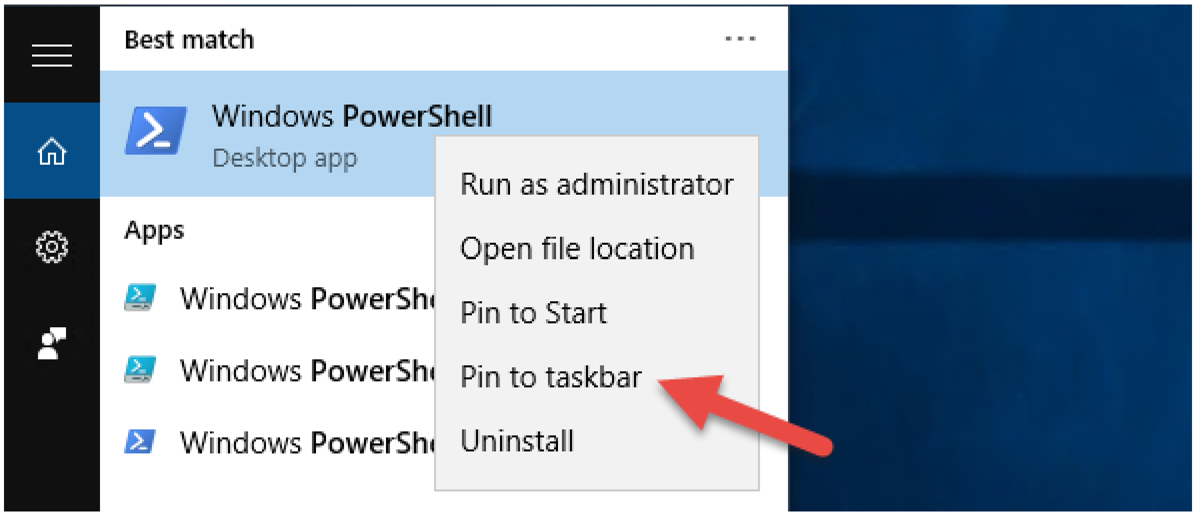


Figure 1-8

Right-click on the PowerShell shortcut that's now pinned to the taskbar and select properties as shown in Figure 1-9.

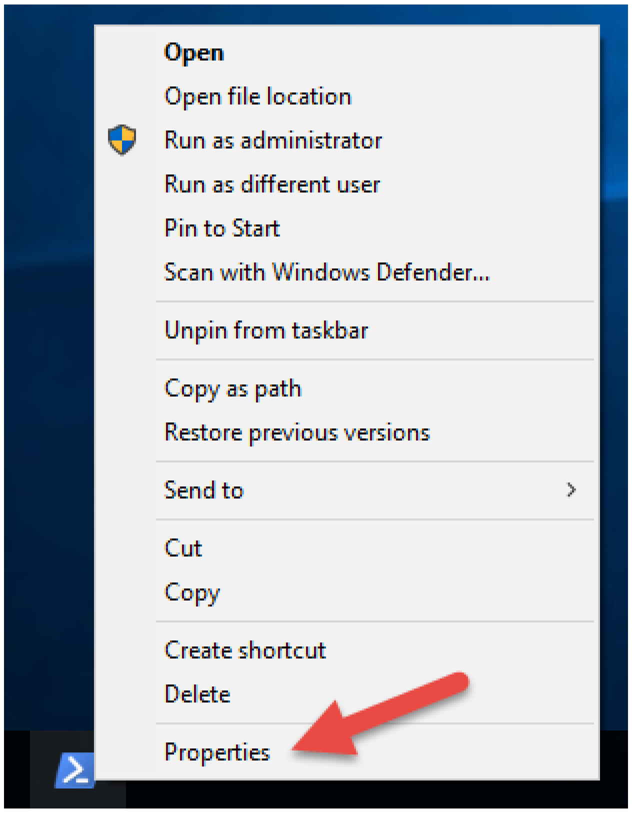


Figure 1-9

Click on "Advanced" as denoted by #1 in Figure 1-10, then check the "Run as administrator" checkbox as denoted by #2 in Figure 1-10, and then click OK twice to accept the changes and exit out of both dialog boxes.

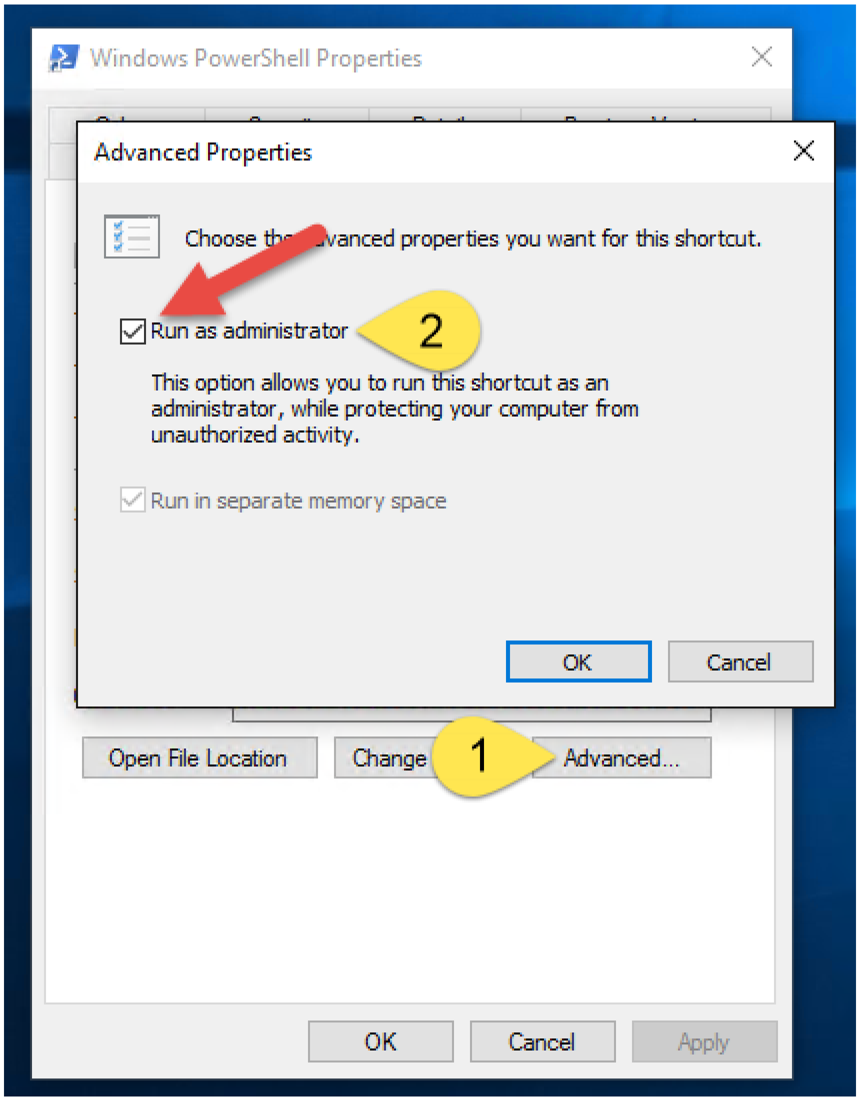

Windows PowerShell Properties
Advanced Properties
Choose the advanced properties you want for this shortcut.
Run as administrator
2
This option allows you to run this shortcut as an administrator, while protecting your computer from unauthorized activity.
Run in separate memory space
OK
Cancel
Open File Location
Change
1
Advanced...
OK
Cancel
Apply

Figure 1-10

You'll never have to worry about finding PowerShell or whether or not it's running as an administrator again.

Running PowerShell elevated as an administrator to prevent having problems with UAC only impacts commands that are run against the local computer. It has no effect on commands that target remote computers.

What version of PowerShell am I running?

There are a number of automatic variables in PowerShell that store state information. One of these variables is $PSVersionTable which contains a hash table that can be used to display the relevant

PowerShell version information:

```
PS C:\> $PSVersionTable
```

```
Name                           Value
----                           -----
PSVersion                      5.1.14393.187
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.14393.187
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1

PS C:\>
```

If $PSVersionTable returns nothing at all, then you have PowerShell version 1.0 and you'll need to update the version of PowerShell that's installed on your computer. Newer versions of PowerShell are distributed as part of the WMF (Windows Management Framework). A specific version of the .NET Framework is required depending on the WMF version. Also note that the full version of the .NET Framework is required and the client version is not sufficient. When updating PowerShell, read the release notes and verify your system meets the requirements. Always check with your application vendors and be sure to test new software (to include a new version of PowerShell) in a test environment before installing it on a production system.

Many times vendors will tell you not to install a new version of PowerShell on the servers that their software runs on. Usually, they don't care what version of PowerShell it's running and what they really mean is that they don't want a newer version of the .NET Framework which is required by the newer version of PowerShell to be installed on the servers that their software is installed on.

Execution Policy

Contrary to popular belief, the execution policy in PowerShell is not a security boundary. It's designed to prevent a user from unknowingly running a script. A determined user can easily bypass the execution policy in PowerShell. Table 1-2 shows the default execution policy for all supported Windows operating systems.

Windows Operating System Version	Default Execution Policy
Windows Vista	Restricted
Server 2008	Restricted
Windows 7	Restricted
Server 2008 R2	Restricted
Windows 8	Restricted
Server 2012	Restricted
Windows 8.1	Restricted
Server 2012 R2	Remote Signed
Windows 10	Restricted
Server 2016	Remote Signed

Table 1-2

Regardless of the execution policy setting, any PowerShell command can be run interactively. The execution policy only comes into play once commands are saved as a script and the script itself is run. The Get-ExecutionPolicy cmdlet is used to determine what the current execution policy setting is and the Set-ExecutionPolicy cmdlet is used to change the execution policy. My recommendation is to use the RemoteSigned policy which requires downloaded scripts to be signed by a trusted publisher in order to be run.

Check the current execution policy:

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

```
Restricted
PS C:\>
```

PowerShell scripts cannot be run at all when the execution policy is set to Restricted which is the default setting on all Windows client operating systems. To demonstrate this, launch the PowerShell ISE (Integrated Scripting Environment) by simply typing ise into the PowerShell console and pressing enter.

Save the following code as a ps1 file named Stop-TimeService.ps1.

```
Get-Service -Name W32Time | Stop-Service -PassThru
```

That command will run interactively without error as long as PowerShell is run elevated as an administrator, but as soon as it's saved as a ps1 (PowerShell script) file and you try to execute the script, it generates an error:

```
PS C:\demo> .\Stop-TimeService.ps1
```

```
.\Stop-TimeService.ps1 : File C:\demo\Stop-TimeService.ps1 cannot be loaded because
running scripts is disabled on this system. For more information, see
about_Execution_Policies at http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\Stop-TimeService.ps1
+
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
PS C:\demo>
```

Notice that the error shown in the previous set of results tells you exactly what the problem is (running scripts is disabled on this system). When you run a command in PowerShell that generates an error message, be sure to read the error message instead of just rerunning the command and hoping that it runs successfully.

Change the PowerShell execution policy to remote signed.

```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned


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

Be sure to read the warning that's displayed when changing the execution policy. I also recommend taking a look at the about_Execution_Policies help topic that's referenced in the warning message shown in the previous set of results, to make sure you understand the security implications of changing the execution policy.

Now that the execution policy has been set to RemoteSigned, the Stop-TimeService.ps1 script runs error free.

```
PS C:\demo> .\Stop-TimeService.ps1
```

```
Status   Name               DisplayName
------   ----               -----------
Stopped  W32Time            Windows Time


PS C:\demo>
```

Be sure to start your Windows Time service before continuing otherwise you may run into unforeseen problems.

```
PS C:\> Start-Service -Name w32time
```

Summary

In this chapter, you've learned where to find and how to launch PowerShell along with how to create a shortcut that will always launch PowerShell as an administrator. You've also learned what versions of PowerShell are supported on which operating systems, what the default execution policy is for each supported operating system, and how to change the execution policy.

Review

1. How do you determine what PowerShell version a computer is running?
2. Why is it important to launch PowerShell elevated as an administrator?
3. How do you determine the current PowerShell execution policy?
4. What does the default PowerShell execution policy on Windows client computers prevent from occurring?
5. How do you change the PowerShell execution policy?

Recommended Reading

For those who want to know more information about the topics covered in this chapter, I recommend reading the following PowerShell help topics.

- about_Automatic_Variables
- about_Execution_Policies

In the next chapter, you'll learn about the discoverability of commands in PowerShell. One of the things that will be covered is how to update PowerShell so those help topics can be viewed right from within PowerShell instead of having to view them on the Internet.

Chapter 2 - The Help System

Two groups of IT Pros were given a written test without access to a computer to determine their skill level with PowerShell. PowerShell beginners were placed in one group and experts in another. Based on the results of the test, there didn't seem to be much difference in the skill level between the two groups. Both groups were given a second test which was similar to the first one except this time they were given access to an isolated computer with PowerShell on it that didn't have access to the Internet. The results of the second test showed there was a huge difference in the skill level of the two groups. Why were these differences observed in the results of the first and second test between these two groups?

Experts don't always know the answers, but they know how to figure out the answers.

The differences in the two tests mentioned in the previous scenario were observed because experts don't memorize how to use thousands of commands in PowerShell. They learn how to use the help system within PowerShell extremely well. This allows them to not only find the necessary commands when needed, but to also figure out how to use those commands once they've found them.

I've heard Jeffrey Snover, the inventor of PowerShell, tell a similar story a number of times.

Mastering the help system is the key to being successful with PowerShell.

Discoverability

Compiled commands in PowerShell are called cmdlets. Cmdlet is pronounced command-let (not CMD-let). Cmdlets are in the form of singular Verb-Noun commands which makes them easily discoverable. For example, the cmdlet for determining what processes are running is Get-Process and the cmdlet for retrieving a list of services and their statuses is Get-Service. There are other types of commands in PowerShell such as aliases and functions which will be covered later in this book. The term PowerShell command is a generic term that's often used to refer to any type of command in PowerShell, regardless of whether or not it's a cmdlet, function, or alias.

The Three Core Cmdlets in PowerShell

- Get-Command
- Get-Help
- Get-Member (Covered in chapter 3)

One question I'm often asked is how do you figure out what the commands are in PowerShell? Both Get-Command and Get-Help can be used to determine the commands.

Get-Help

Get-Help is a multipurpose command. Get-Help helps you learn how to use commands once you find them. Get-Help can also be used to help locate commands, but in a different and more indirect way when compared to Get-Command.

When Get-Help is used to locate commands, it first searches for wildcard matches of command names based on the provided input. If it doesn't find a match, it searches through the help topics themselves, and if no match is found an error is returned. Contrary to popular belief, Get-Help can be used to find commands that don't have help topics.

The first thing you need to know about the help system in PowerShell is how to use the Get-Help cmdlet. The following command is used to display the help topic for Get-Help.

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

```
Do you want to run Update-Help?
The Update-Help cmdlet downloads the most current Help files for Windows PowerShell
modules, and installs them on your computer. For more information about the Update-Help
cmdlet, see http://go.microsoft.com/fwlink/?LinkId=210614.
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):
```

Beginning with PowerShell version 3, PowerShell help doesn't ship with the operating system, so the first time Get-Help is run for a command, the previous message will be displayed. If the help function or man alias is used instead of the Get-Help cmdlet, you won't receive this prompt.

Answering yes by pressing "Y" (Y can be specified in either upper or lower case), runs the Update-Help cmdlet which requires Internet access by default.

Once the help is downloaded and the update is complete, the help topic is returned for the specified command:

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

```
Do you want to run Update-Help?
The Update-Help cmdlet downloads the most current Help files for Windows PowerShell
modules, and installs them on your computer. For more information about the Update-Help
cmdlet, see http://go.microsoft.com/fwlink/?LinkId=210614.
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y

NAME
    Get-Help

SYNOPSIS
    Displays information about Windows PowerShell commands and concepts.


SYNTAX
    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Full] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>]
    [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>]
    -Detailed [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>]
    -Examples [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Online
    [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>]
    -Parameter <String> [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>]
    -ShowWindow [<CommonParameters>]


DESCRIPTION
    The Get-Help cmdlet displays information about Windows PowerShell concepts and
    commands, including cmdlets, functions, CIM commands, workflows, providers, aliases
    and scripts.

    To get help for a Windows PowerShell command, type Get-Help followed by the command
    name, such as: Get-Help Get-Process. To get a list of all help topics on your
    system, type Get-Help *. You can display the whole help topic or use the parameters
    of the Get-Help cmdlet to get selected parts of the topic, such as the syntax,
    parameters, or examples.

    Conceptual help topics in Windows PowerShell begin with "about_", such as
    "about_Comparison_Operators". To see all "about_" topics, type Get-Help about_*. To
    see a particular topic, type Get-Help about_<topic-name>, such as Get-Help
    about_Comparison_Operators.

    To get help for a Windows PowerShell provider, type Get-Help followed by the
    provider name. For example, to get help for the Certificate provider, type Get-Help
    Certificate.

    In addition to Get-Help, you can also type help or man, which displays one screen of
    text at a time, or <cmdlet-name> -?, which is identical to Get-Help but works only
    for commands.

    Get-Help gets the help content that it displays from help files on your computer.
    Without the help files, Get-Help displays only basic information about commands.
    Some Windows PowerShell modules come with help files. However, starting in Windows
    PowerShell 3.0, the modules that come with the Windows operating system do not
    include help files. To download or update the help files for a module in Windows
    PowerShell 3.0, use the Update-Help cmdlet.

    You can also view the help topics for Windows PowerShell online in the TechNet
    Library. To get the online version of a help topic, use the Online parameter, such
    as: Get-Help Get-Process -Online. To read all of the help topics, see Scripting with
    Windows PowerShell (http://go.microsoft.com/fwlink/?LinkID=107116) in the TechNet
    library.

    If you type Get-Help followed by the exact name of a help topic, or by a word unique
    to a help topic, Get-Help displays the topic contents. If you enter a word or word
    pattern that appears in several help topic titles, Get-Help displays a list of the
    matching titles. If you enter a word that does not appear in any help topic titles,
    Get-Help displays a list of topics that include that word in their contents.
    Get-Help can get help topics for all supported languages and locales. Get-Help first
    looks for help files in the locale set for Windows, then in the parent locale, such
    as "pt" for "pt-BR", and then in a fallback locale. Beginning in Windows PowerShell
    3.0, if Get-Help does not find help in the fallback locale, it looks for help topics
    in English, "en-US", before it returns an error message or displaying auto-generated
    help.

    For information about the symbols that Get-Help displays in the command syntax
    diagram, see about_Command_Syntax. For information about parameter attributes, such
    as Required and Position, see about_Parameters.

    TROUBLESHOOTING NOTE: In Windows PowerShell 3.0 and Windows PowerShell 4.0, Get-Help
    cannot find About topics in modules unless the module is imported into the current
    session. This is a known issue. To get About topics in a module, import the module,
    either by using the Import-Module cmdlet or by running a cmdlet in the module.


RELATED LINKS
    Online Version: http://go.microsoft.com/fwlink/?LinkId=821483
    Updatable Help Status Table (http://go.microsoft.com/fwlink/?LinkID=270007)
    about_Command_Syntax
    Get-Command
    about_Comment_Based_Help
    about_Parameters

REMARKS
    To see the examples, type: "get-help Get-Help -examples".
    For more information, type: "get-help Get-Help -detailed".
    For technical information, type: "get-help Get-Help -full".
    For online help, type: "get-help Get-Help -online"

PS C:\>
```

As you can see, help topics can contain an enormous amount of information and this isn't even the entire help topic.

While not specific to PowerShell, a parameter is a way to provide input to a command. Get-Help has numerous parameters that can be specified in order to return the entire help topic or a subset of it.

The syntax section of the help topic shown in the previous set of results lists all of the parameters for Get-Help. At first glance, it appears the same parameters are listed six different times. Each of those different blocks in the syntax section is a parameter set. This means the Get-Help cmdlet has six different parameter sets. If you take a closer look, you'll notice that at least one parameter is different in each of the parameter sets.

Parameter sets are mutually exclusive. Once a unique parameter that only exists in one of the parameter sets is used, only parameters contained within that parameter set can be used. For example, both the Full and Detailed parameters couldn't be specified at the same time because they reside in different parameter sets.

The following parameters each reside in different parameter sets:

- Full
- Detailed
- Examples
- Online
- Parameter
- ShowWindow

All of the cryptic syntax such as square and angle brackets in the syntax section means something but will be covered in Appendix A of this book. While important, learning what the cryptic syntax means is often short lived since it's difficult to retain for someone who is new to PowerShell and may not use it everyday. For beginners, there's an easier way to figure out the same information except in plain English.

When the Full parameter of Get-Help is specified, the entire help topic is returned.

```
PS C:\> Get-Help -Name Get-Help -Full
```

```
NAME
    Get-Help

SYNOPSIS
    Displays information about Windows PowerShell commands and concepts.


SYNTAX
    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] -Detailed [-Functionality <String[]>] [-Path <String>] [-Role
    <String[]>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] -Examples [-Functionality <String[]>] [-Path <String>] [-Role
    <String[]>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Full] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>]
    [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Functionality <String[]>] -Online [-Path <String>] [-Role <String[]>]
    [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Functionality <String[]>] -Parameter <String> [-Path <String>] [-Role
    <String[]>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category {Alias | Cmdlet | Provider | General | FAQ |
    Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All |
    DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component
    <String[]>] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>]
    -ShowWindow [<CommonParameters>]


DESCRIPTION
    The Get-Help cmdlet displays information about Windows PowerShell concepts and
    commands, including cmdlets, functions, CIM commands, workflows, providers, aliases
    and scripts.

    To get help for a Windows PowerShell command, type `Get-Help` followed by the
    command name, such as: `Get-Help Get-Process`. To get a list of all help topics on
    your system, type `Get-Help *`. You can display the whole help topic or use the
    parameters of the Get-Help cmdlet to get selected parts of the topic, such as the
    syntax, parameters, or examples.

    Conceptual help topics in Windows PowerShell begin with "about_", such as
    "about_Comparison_Operators". To see all "about_" topics, type `Get-Help about_*`.
    To see a particular topic, type `Get-Help about_<topic-name>`, such as `Get-Help
    about_Comparison_Operators`.

    To get help for a Windows PowerShell provider, type `Get-Help` followed by the
    provider name. For example, to get help for the Certificate provider, type `Get-Help
    Certificate`.

    In addition to `Get-Help`, you can also type `help` or `man`, which displays one
    screen of text at a time, or `<cmdlet-name> -?`, which is identical to Get-Help but
    works only for commands. Get-Help gets the help content that it displays from help
    files on your computer. Without the help files, Get-Help displays only basic
    information about commands. Some Windows PowerShell modules come with help files.
    However, starting in Windows PowerShell 3.0, the modules that come with the Windows
    operating system do not include help files. To download or update the help files for
    a module in Windows PowerShell 3.0, use the Update-Help cmdlet.

    You can also view the help topics for Windows PowerShell online in the TechNet
    Library. To get the online version of a help topic, use the Online parameter, such
    as: `Get-Help Get-Process -Online`. To read all of the help topics, see Scripting
    with Windows PowerShellhttp://go.microsoft.com/fwlink/?LinkID=107116
    (http://go.microsoft.com/fwlink/?LinkID=107116) in the TechNet library.

    If you type `Get-Help` followed by the exact name of a help topic, or by a word
    unique to a help topic, Get-Help displays the topic contents. If you enter a word or
    word pattern that appears in several help topic titles, Get-Help displays a list of
    the matching titles. If you enter a word that does not appear in any help topic
    titles, Get-Help displays a list of topics that include that word in their contents.
    Get-Help can get help topics for all supported languages and locales. Get-Help first
    looks for help files in the locale set for Windows, then in the parent locale, such
    as "pt" for "pt-BR", and then in a fallback locale. Beginning in Windows PowerShell
    3.0, if Get-Help does not find help in the fallback locale, it looks for help topics
    in English, "en-US", before it returns an error message or displaying auto-generated
    help.

    For information about the symbols that Get-Help displays in the command syntax
    diagram, see about_Command_Syntax. For information about parameter attributes, such
    as Required and Position , see about_Parameters. TROUBLESHOOTING NOTE : In Windows
    PowerShell 3.0 and Windows PowerShell 4.0, Get-Help cannot find About topics in
    modules unless the module is imported into the current session. This is a known
    issue. To get About topics in a module, import the module, either by using the
    Import-Module cmdlet or by running a cmdlet in the module.


PARAMETERS
    -Category <String[]>
        Displays help only for items in the specified category and their aliases. The
        acceptable values for this parameter are:

        - Alias

        - Cmdlet

        - Provider

        - General

        - FAQ

        - Glossary
        - HelpFile

        - ScriptCommand

        - Function

        - Filter

        - ExternalScript

        - All

        - DefaultHelp

        - Workflow

        - DscResource

        - Class

        - Configuration


        Conceptual topics are in the HelpFile category.


        Required?                    false
        Position?                    named
        Default value                None
        Accept pipeline input?       False
        Accept wildcard characters?  false

    -Component <String[]>
        Displays commands with the specified component value, such as "Exchange." Enter
        a component name. Wildcard characters are permitted.

        This parameter has no effect on displays of conceptual ("About_") help.

        Required?                    false
        Position?                    named
        Default value                None
        Accept pipeline input?       False
        Accept wildcard characters?  false

    -Detailed [<SwitchParameter>]
        Adds parameter descriptions and examples to the basic help display.
        This parameter is effective only when help files are for the command are
        installed on the computer. It has no effect on displays of conceptual ("About_")
        help.

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

    -Examples [<SwitchParameter>]
        Displays only the name, synopsis, and examples. To display only the examples,
        type `(Get-Help <cmdlet-name>).Examples`.

        This parameter is effective only when help files are for the command are
        installed on the computer. It has no effect on displays of conceptual ("About_")
        help.

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

    -Full [<SwitchParameter>]
        Displays the whole help topic for a cmdlet. This includes parameter descriptions
        and attributes, examples, input and output object types, and additional notes.

        This parameter is effective only when help files are for the command are
        installed on the computer. It has no effect on displays of conceptual ("About_")
        help.

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

    -Functionality <String[]>
        Displays help for items with the specified functionality. Enter the
        functionality. Wildcard characters are permitted.

        This parameter has no effect on displays of conceptual ("About_") help.

        Required?                    false
        Position?                    named
        Default value                None
        Accept pipeline input?       False
        Accept wildcard characters?  false

    -Name <String>
        Gets help about the specified command or concept. Enter the name of a cmdlet,
        function, provider, script, or workflow, such as `Get-Member`, a conceptual
        topic name, such as `about_Objects`, or an alias, such as `ls`. Wildcard
        characters are permitted in cmdlet and provider names, but you cannot use
        wildcard characters to find the names of function help and script help topics.

        To get help for a script that is not located in a path that is listed in the
        Path environment variable, type the path and file name of the script.

        If you enter the exact name of a help topic, Get-Help displays the topic
        contents. If you enter a word or word pattern that appears in several help topic
        titles, Get-Help displays a list of the matching titles. If you enter a word
        that does not match any help topic titles, Get-Help displays a list of topics
        that include that word in their contents.

        The names of conceptual topics, such as `about_Objects`, must be entered in
        English, even in non-English versions of Windows PowerShell.

        Required?                    false
        Position?                    0
        Default value                None
        Accept pipeline input?       True (ByPropertyName)
        Accept wildcard characters?  false

    -Online [<SwitchParameter>]
        Displays the online version of a help topic in the default Internet browser.
        This parameter is valid only for cmdlet, function, workflow and script help
        topics. You cannot use the Online parameter in Get-Help commands in a remote
        session.

        For information about supporting this feature in help topics that you write, see
        about_Comment_Based_Help (http://go.microsoft.com/fwlink/?LinkID=144309), and
        Supporting Online Help (http://go.microsoft.com/fwlink/?LinkID=242132), and How
        to Write Cmdlet Helphttp://go.microsoft.com/fwlink/?LinkID=123415
        (http://go.microsoft.com/fwlink/?LinkID=123415) in the Microsoft Developer
        Network MSDN library.

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

```
-Parameter <String>
    Displays only the detailed descriptions of the specified parameters. Wildcards
    are permitted.

    This parameter has no effect on displays of conceptual ("About_") help.

    Required?                    true
    Position?                    named
    Default value                None
    Accept pipeline input?       False
    Accept wildcard characters?  false

-Path <String>
    Gets help that explains how the cmdlet works in the specified provider path.
    Enter a Windows PowerShell provider path.

    This parameter gets a customized version of a cmdlet help topic that explains
    how the cmdlet works in the specified Windows PowerShell provider path. This
    parameter is effective only for help about a provider cmdlet and only when the
    provider includes a custom version of the provider cmdlet help topic in its help
    file. To use this parameter, install the help file for the module that includes
    the provider.

    To see the custom cmdlet help for a provider path, go to the provider path
    location and enter a Get-Help command or, from any path location, use the Path
    parameter of Get-Help to specify the provider path. You can also find custom
    cmdlet help online in the provider help section of the help topics. For example,
    you can find help for the New-Item cmdlet in the Wsman:\*\ClientCertificate path
    (http://go.microsoft.com/fwlink/?LinkID=158676).

    For more information about Windows PowerShell providers, see about_Providers
    (http://go.microsoft.com/fwlink/?LinkID=113250) in the TechNet library.

    Required?                    false
    Position?                    named
    Default value                None
    Accept pipeline input?       False
    Accept wildcard characters?  false

-Role <String[]>
    Displays help customized for the specified user role. Enter a role. Wildcard
    characters are permitted.

    Enter the role that the user plays in an organization. Some cmdlets display
    different text in their help files based on the value of this parameter. This
    parameter has no effect on help for the core cmdlets.
```

```
        Required?                    false
        Position?                    named
        Default value                None
        Accept pipeline input?       False
        Accept wildcard characters?  false

    -ShowWindow [<SwitchParameter>]
        Displays the help topic in a window for easier reading. The window includes a
        Find search feature and a Settings box that lets you set options for the
        display. These include options to display only selected sections of a help topic.

        The ShowWindow parameter supports help topics for commands, which include
        cmdlets, functions, CIM commands, workflows, and scripts, and conceptual About
        topics. It does not support provider help.

        This parameter was introduced in Windows PowerShell 3.0.

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

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    None
        You cannot pipe objects to this cmdlet.


OUTPUTS
    ExtendedCmdletHelpInfo
        If you run Get-Help on a command that does not have a help file, Get-Help
        returns an ExtendedCmdletHelpInfo object that represents autogenerated help.

    System.String
        If you get a conceptual help topic, Get-Help returns it as a string.

    MamlCommandHelpInfo
        If you get a command that has a help file, Get-Help returns a
        MamlCommandHelpInfo object.
```

```
NOTES


    If you do not specify parameters, Get-Help * displays information about the
    Windows PowerShell help system. Windows PowerShell 3.0 does not include help
    files. To download and install the help files that Get-Help reads, use the
    Update-Help cmdlet. You can use the Update-Help * cmdlet to download and install
    help files for the core commands that come with Windows PowerShell and for any
    modules that you install. You can also use it to update the help files so that
    the help on your computer is never outdated.

    You can also read the help topics about the commands that come with Windows
    PowerShell online starting at Scripting with Windows
    PowerShellhttp://go.microsoft.com/fwlink/?LinkID=107116
    (http://go.microsoft.com/fwlink/?LinkID=107116).  Get-Help displays help in the
    locale set for the Windows operating system or in the fallback language for that
    locale. If you do not have help files for the primary or fallback locale,
    Get-Help * behaves as if there are no help files on the computer. To get help
    for a different locale, use Region and Language in Control Panel to change the
    settings.

    * The full view of help includes a table of information about the parameters.
    The table includes the following fields:


    - Required. Indicates whether the parameter is required (true) or optional
    (false).

    - Position. Indicates whether the parameter is named or positional (numbered).
    Positional parameters must appear in a specified place in the command.

    ---- "Named" indicates that the parameter name is required, but that the
    parameter can appear anywhere in the command.

    ---- <Number> indicates that the parameter name is optional, but when the name
    is omitted, the parameter must be in the place specified by the number. For
    example, "2" indicates that when the parameter name is omitted, the parameter
    must be the second (2) or only unnamed parameter in the command. When the
    parameter name is used, the parameter can appear anywhere in the command.

    - Default value. The parameter value that Windows PowerShell uses if you do not
    include the parameter in the command.

    - Accepts pipeline input. Indicates whether you can (true) or cannot (false)
    send objects to the parameter through a pipeline. "By Property Name" means that
    the pipelined object must have a property that has the same name as the
    parameter name.
    - Accepts wildcard characters. Indicates whether the value of a parameter can
    include wildcard characters, such as * and ?.


Example 1: Display help about the help system

PS C:\>Get-Help

This command displays help about the Windows PowerShell help system.
Example 2: Display available help topics

PS C:\>Get-Help *

This command displays a list of the available help topics.
Example 3: Display basic information about a cmdlet

PS C:\>Get-Help Get-Alias
PS C:\> Help Get-Alias
PS C:\> Get-Alias -?

These commands display basic information about the Get-Alias cmdlet. The Get-Help
and ? commands display the information on a single page. The Help command displays
the information one page at a time.
Example 4: Display a list of conceptual topics

PS C:\>Get-Help about_*

This command displays a list of the conceptual topics included in Windows PowerShell
help. All of these topics begin with the characters about_. To display a particular
help file, type get-help <topic-name>, for example, `Get-Help about_Signing`.

This command displays the conceptual topics only when the help files for those
topics are installed on the computer. For information about downloading and
installing help files in Windows PowerShell 3.0, see Update-Help.
Example 5: Download and install help files

The first command uses the **Get-Help** cmdlet to get help for the Get-Command
cmdlet. Without help files, **Get-Help** display the cmdlet name, syntax and alias
of **Get-Command**, and prompts you to use the **Update-Help** cmdlet to get the
newest help files.
PS C:\>Get-Help Get-Command
NAME
    Get-Command


SYNTAX
```

```
 Get-Command [[-Name] <string[]>] [-CommandType {Alias | Function | Filter | Cmdlet
| ExternalScript | Application |

    Script | All}] [[-ArgumentList] <Object[]>] [-Module <string[]>] [-Syntax]
[-TotalCount <int>] [<CommonParameters>]


    Get-Command [-Noun <string[]>] [-Verb <string[]>] [[-ArgumentList] <Object[]>]
[-Module <string[]>] [-Syntax]

    [-TotalCount <int>] [<CommonParameters>]

ALIASES
    gcm


REMARKS
    Get-Help cannot find the help files for this cmdlet on this computer.
    It is displaying only partial help. To download and install help files
    for this cmdlet, use **Update-Help**.

The second command runs the **Update-Help** cmdlet without parameters. This command
downloads help files from the Internet for all of the modules in the current session
and installs them on the local computer.This command works only when the local
computer is connected to the Internet. If your computer is not connected to the
Internet, you might be able to install help files from a network share. For more
information, see Save-Help.
PS C:\>Update-Help

Now that the help files are downloaded, we can repeat the first command in the
sequence. This command gets help for the **Get-Command** cmdlet. The cmdlet now gets
more extensive help for **Get-Command** and you can use the *Detailed*, *Full*,
*Example*, and *Parameter* parameters of **Get-Help** to customize the displays.You
can use the **Get-Help** cmdlet as soon as the **Update-Help** command finishes. You
do not have to restart Windows PowerShell.
PS C:\>Get-Help Get-Command

This example shows how to download and install new or updated help files for a
module. It uses features that were introduced in Windows PowerShell 3.0.

The example compares the help that Get-Help displays for commands when you do not
have help files installed on your computer and when you do have help files. You can
use the same command sequence to update the help files on your computer so that your
local help content is never obsolete.

To download and install the help files for the commands that come with Windows
```

```
PowerShell, and for any modules in the $pshome\Modules directory, open Windows
PowerShell by using the Run as administrator option. If you are not a member of the
Administrators group on the computer, you cannot download help for these modules.
However, you can use the Online parameter to open the online version of help for a
command, and you can read the help for Windows PowerShell in the TechNet library
starting at Scripting with Windows
PowerShellhttp://go.microsoft.com/fwlink/?LinkID=107116
(http://go.microsoft.com/fwlink/?LinkID=107116).
Example 6: Display detailed help

PS C:\>Get-Help ls -Detailed

This command displays detailed help for the Get-ChildItem cmdlet by specifying one
of its aliases, ls. The Detailed parameter of Get-Help gets the detailed view of the
help topic, which includes parameter descriptions and examples. To see the complete
help topic for a cmdlet, use the Full parameter.

The Full and Detailed parameters are effective only when help files for the command
are installed on the computer.
Example 7: Display full information for a cmdlet

PS C:\>Get-Help Format-Table -Full

This command uses the Full parameter of Get-Help to display the full view help for
the Format-Table cmdlet. The full view of help includes parameter descriptions,
examples, and a table of technical details about the parameters.

The Full parameter is effective only when help files for the command are installed
on the computer.
Example 8: Display examples for a cmdlet

PS C:\>Get-Help Start-Service -Examples

This command displays examples of using the Start-Service cmdlet. It uses the
Examples parameter of Get-Help to display only the Examples section of the cmdlet
help topics.

The Examples parameter is effective only when help files for the command are
installed on the computer.
Example 9: Display parameter help

PS C:\>Get-Help Format-List -Parameter GroupBy

This command uses the Parameter parameter of Get-Help to display a detailed
description of the GroupBy parameter of the Format-List cmdlet. For detailed
descriptions of all parameters of the Format-List cmdlet, type `Get-Help Format-List
-Parameter *`.
```

```
Example 10: Search for a word in cmdlet help

PS C:\>Get-Help Add-Member -Full | Out-String -Stream | Select-String -Pattern Clixml

This example shows how to search for a word in particular cmdlet help topic. This
command searches for the word Clixml in the full version of the help topic for the
Add-Member cmdlet.

Because the Get-Help cmdlet generates a MamlCommandHelpInfo object, not a string,
you have to use a cmdlet that transforms the help topic content into a string, such
as Out-String or Out-File.
Example 11: Display online version of help

PS C:\>Get-Help Get-Member -Online

This command displays the online version of the help topic for the Get-Member cmdlet.
Example 12: Display a list of topics that include a word

PS C:\>Get-Help remoting

This command displays a list of topics that include the word remoting.

When you enter a word that does not appear in any topic title, Get-Help displays a
list of topics that include that word.
Example 13: Display provider specific help

The first command uses the *Path* parameter of **Get-Help** to specify the provider
path. This command can be entered at any path location.
PS C:\>Get-Help Get-Item -Path SQLSERVER:\DataCollection

NAME

    Get-Item


SYNOPSIS

    Gets a collection of Server objects for the local computer and any computers

    to which you have made a SQL Server PowerShell connection.

    ...

The second command uses the Set-Location cmdlet (alias = "cd") to navigate to the
provider path. From that location, even without the *Path* parameter, the
**Get-Help** command gets the provider-specific help for the **Get-Item** cmdlet.
PS C:\>cd SQLSERVER:\DataCollection
```

```
    SQLSERVER:\DataCollection> Get-Help Get-Item

    NAME

        Get-Item


    SYNOPSIS

        Gets a collection of Server objects for the local computer and any computers

        to which you have made a SQL Server PowerShell connection.

        ...


    The third command shows that a **Get-Help** command in a file system path, and
    without the *Path* parameter, gets the standard help for the **Get-Item** cmdlet.
    PS C:\>Get-Item

    NAME

        Get-Item


    SYNOPSIS

        Gets the item at the specified location.
        ...

    This example shows how to get help that explains how to use the Get-Item cmdlet in
    the DataCollection node of the Windows PowerShellSQL Server provider. The example
    shows two ways of getting the provider-specific help for Get-Item .

    You can also get provider-specific help for cmdlets online in the section that
    describes the provider. For example, for provider-specific online help for the
    New-Item cmdlet in each WSMan provider path, see New-Item for
    ClientCertificatehttp://go.microsoft.com/fwlink/?LinkID=158676 in the TechNet
    library.
    Example 14: Display help for a script

    PS C:\>Get-Help C:\PS-Test\MyScript.ps1

    This command gets help for the MyScript.ps1 script. For information about how to
    write help for your functions and scripts, see about_Comment_Based_Help.

RELATED LINKS
```

```
    Online Version: http://go.microsoft.com/fwlink/?LinkId=821483
    Updatable Help Status Table (http://go.microsoft.com/fwlink/?LinkID=270007)
    http://go.microsoft.com/fwlink/?LinkID=270007
    about_Command_Syntax
    Get-Command

PS C:\>
```

Notice that using the Full parameter returned several additional sections, one of which is the Parameters section which provides the same information and more, as the cryptic syntax in the Syntax section except in plain English.

The Full parameter is a switch parameter. A parameter that doesn't require a value is called a switch parameter. When a switch parameter is specified, it's on and when it's not, it's off.

If you've been working through this chapter in the PowerShell console of your Windows 10 lab environment computer, you noticed that the previous command to display the full help topic for Get-Help flew by on the screen without giving you a chance to read it. There's a better way.

Help is a function that pipes Get-Help to a function named More which is a wrapper for the More.com executable file in Windows. In the PowerShell console, Help provides a page of help at a time but it works the same way as Get-Help in the ISE (Integrated Scripting Environment). My recommendation is to use the Help function instead of the Get-Help cmdlet since it provides a better experience and it's less to type.

Less typing isn't always a good thing, however. If you're going to save your commands as a script or share them with someone else, be sure to use full cmdlet and parameter names since they're more self-documenting which makes them easier to understand. Think about the next person that has to read and understand your commands, it could be you. Your co-workers and future self will thank you.

Try running the following commands in the PowerShell console on your Windows 10 lab environment computer.

- Get-Help -Name Get-Help -Full
- help -Name Get-Help -Full
- help Get-Help -Full

Did you notice any differences in the output from the previously listed commands when you ran them on your Windows 10 lab environment computer?

There aren't any differences other than the last two options return the results one page at a time. The spacebar is used to display the next page of content when using the Help function and Ctrl+C cancels commands that are running in the PowerShell console.

The first example uses the Get-Help cmdlet, the second uses the Help function, and the third omits the Name parameter when using the Help function. Name is a positional parameter and it's being used positionally in that example. This means the value can be specified without specifying the parameter name, as long as the value itself is specified in the correct position. How did I know what position to specify the value in? By reading the help as shown in the following example.

```
PS C:\> help Get-Help -Parameter Name

-Name <String>
    Gets help about the specified command or concept. Enter the name of a cmdlet,
    function, provider, script, or workflow, such as `Get-Member`, a conceptual topic
    name, such as `about_Objects`, or an alias, such as `ls`. Wildcard characters are
    permitted in cmdlet and provider names, but you cannot use wildcard characters to
    find the names of function help and script help topics.

    To get help for a script that is not located in a path that is listed in the Path
    environment variable, type the path and file name of the script.

    If you enter the exact name of a help topic, Get-Help displays the topic contents.
    If you enter a word or word pattern that appears in several help topic titles,
    Get-Help displays a list of the matching titles. If you enter a word that does not
    match any help topic titles, Get-Help displays a list of topics that include that
    word in their contents.

    The names of conceptual topics, such as `about_Objects`, must be entered in English,
    even in non-English versions of Windows PowerShell.

    Required?                    false
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByPropertyName)
    Accept wildcard characters?  false

PS C:\>
```

Notice that in the previous example, the Parameter parameter was used with the Help function to only return information from the help topic for the Name parameter. This is much more concise than trying to manually sift through what sometimes seems like a hundred page help topic.

Based on those results, you can see that the Name parameter is positional and must be specified in position zero (the first position) when used positionally. The order that parameters are specified in doesn't matter if the parameter name is specified.

One other important piece of information in the previous results is that the Name parameter expects the datatype for its value to be a single string which is denoted by <String>. If it accepted multiple strings, the datatype would be listed as <String[]>.

Sometimes you simply don't want to display the entire help topic for a command. There are a number of other parameters besides Full that can be specified with Get-Help or Help. Try running the following commands on your Windows 10 lab environment computer:

- Get-Help -Name Get-Command -Full
- Get-Help -Name Get-Command -Detailed
- Get-Help -Name Get-Command -Examples
- Get-Help -Name Get-Command -Online
- Get-Help -Name Get-Command -Parameter Noun
- Get-Help -Name Get-Command -ShowWindow

I typically use help <command name> with the Full or Online parameter. If I'm only interested in the examples, I'll use the Examples parameter and if I'm only interested in a specific parameter, I'll use the Parameter parameter. The ShowWindow parameter opens the help topic in a separate searchable window that can be placed on a different monitor if you have multiple monitors. I've avoided the ShowWindow parameter because there's a known bug where it doesn't display the entire help topic.

If you want help in a separate window, my recommendation is to either use the Online parameter or use the Full parameter and pipe the results to Out-GridView, as shown in the following example.

```
help Get-Command -Full | Out-GridView
```

Both the Out-GridView cmdlet and the ShowWindow parameter of the Get-Help cmdlet require an operating system with a GUI (Graphical User Interface). They will generate an error message if you attempt to use either of them on Windows Server that's been installed using the server core (no-GUI) installation option.

To use Get-Help to find commands, use the asterisk (*) wildcard character with the Name parameter. Specify a term that you're searching for commands on as the value for the Name parameter as shown in the following example.

```
PS C:\> help *process*

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
Enter-PSHostProcess               Cmdlet    Microsoft.PowerShell.Core Connects to and ...
Exit-PSHostProcess                Cmdlet    Microsoft.PowerShell.Core Closes an intera...
Get-PSHostProcessInfo             Cmdlet    Microsoft.PowerShell.Core
Debug-Process                     Cmdlet    Microsoft.PowerShell.M... Debugs one or mo...
Get-Process                       Cmdlet    Microsoft.PowerShell.M... Gets the process...
Start-Process                     Cmdlet    Microsoft.PowerShell.M... Starts one or mo...
Stop-Process                      Cmdlet    Microsoft.PowerShell.M... Stops one or mor...
Wait-Process                      Cmdlet    Microsoft.PowerShell.M... Waits for the pr...
Get-AppvVirtualProcess            Function  AppvClient                ...
Start-AppvVirtualProcess          Function  AppvClient                ...

PS C:\>
```

In the previous example, the * wildcard characters are not required and omitting them produces the same result. Get-Help automatically adds the wildcard characters behind the scenes.

```
PS C:\> help process
```

The previous command produces the same results as specifying the * wildcard character on each end of process.

I prefer to add them since that's the option that always works consistently. Otherwise, they are required in certain scenarios and not others. As soon as you add a wildcard character in the middle of the value, they're no longer automatically added behind the scenes to the value you specified.

```
PS C:\> help pr*cess
```

No results are returned by that command unless the * wildcard character is added to the beginning, end, or both the beginning and end of pr*cess.

If the value you specified begins with a dash then an error will be generated because PowerShell interprets it as a parameter name and no such parameter name exists for the Get-Help cmdlet.

```
PS C:\> help -process
```

If what you're attempting to look for are commands that end with -process, you would simply need to add the * wildcard character to the beginning of the value.

```
PS C:\> help *-process
```

When searching for PowerShell commands with Get-Help, you want to be a little more vague instead of being too specific with what you're searching for.

Searching for process earlier found only commands that contained process in the name of the command and returned only those results. When Get-Help is used to search for processes, it doesn't find any matches for command names, so it performs a search of every help topic in PowerShell on your system and returns any matches it finds. This causes it to return an enormous amount of results.

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

```
Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
Disconnect-PSSession              Cmdlet    Microsoft.PowerShell.Core Disconnects from...
Enter-PSHostProcess               Cmdlet    Microsoft.PowerShell.Core Connects to and ...
ForEach-Object                    Cmdlet    Microsoft.PowerShell.Core Performs an oper...
Get-PSSessionConfiguration        Cmdlet    Microsoft.PowerShell.Core Gets the registe...
New-PSTransportOption             Cmdlet    Microsoft.PowerShell.Core Creates an objec...
Out-Host                          Cmdlet    Microsoft.PowerShell.Core Sends output to ...
Where-Object                      Cmdlet    Microsoft.PowerShell.Core Selects objects ...
Clear-Variable                    Cmdlet    Microsoft.PowerShell.U... Deletes the valu...
Compare-Object                    Cmdlet    Microsoft.PowerShell.U... Compares two set...
Convert-String                    Cmdlet    Microsoft.PowerShell.U... Formats a string...
ConvertFrom-Csv                   Cmdlet    Microsoft.PowerShell.U... Converts object ...
ConvertTo-Html                    Cmdlet    Microsoft.PowerShell.U... Converts Microso...
ConvertTo-Xml                     Cmdlet    Microsoft.PowerShell.U... Creates an XML-b...
Debug-Runspace                    Cmdlet    Microsoft.PowerShell.U... Starts an intera...
Export-Csv                        Cmdlet    Microsoft.PowerShell.U... Converts objects...
Export-FormatData                 Cmdlet    Microsoft.PowerShell.U... Saves formatting...
Format-List                       Cmdlet    Microsoft.PowerShell.U... Formats the outp...
Format-Table                      Cmdlet    Microsoft.PowerShell.U... Formats the outp...
Get-Random                        Cmdlet    Microsoft.PowerShell.U... Gets a random nu...
Get-Unique                        Cmdlet    Microsoft.PowerShell.U... Returns unique i...
Group-Object                      Cmdlet    Microsoft.PowerShell.U... Groups objects t...
Import-Clixml                     Cmdlet    Microsoft.PowerShell.U... Imports a CLIXML...
Import-Csv                        Cmdlet    Microsoft.PowerShell.U... Creates table-li...
Measure-Object                    Cmdlet    Microsoft.PowerShell.U... Calculates the n...
Out-File                          Cmdlet    Microsoft.PowerShell.U... Sends output to ...
Out-GridView                      Cmdlet    Microsoft.PowerShell.U... Sends output to ...
Select-Object                     Cmdlet    Microsoft.PowerShell.U... Selects objects ...
Set-Variable                      Cmdlet    Microsoft.PowerShell.U... Sets the value o...
Sort-Object                       Cmdlet    Microsoft.PowerShell.U... Sorts objects by...
Tee-Object                        Cmdlet    Microsoft.PowerShell.U... Saves command ou...
Trace-Command                     Cmdlet    Microsoft.PowerShell.U... Configures and s...
Write-Output                      Cmdlet    Microsoft.PowerShell.U... Sends the specif...
Debug-Process                     Cmdlet    Microsoft.PowerShell.M... Debugs one or mo...
Get-Process                       Cmdlet    Microsoft.PowerShell.M... Gets the process...
Get-WmiObject                     Cmdlet    Microsoft.PowerShell.M... Gets instances o...
Start-Process                     Cmdlet    Microsoft.PowerShell.M... Starts one or mo...
Stop-Process                      Cmdlet    Microsoft.PowerShell.M... Stops one or mor...
Wait-Process                      Cmdlet    Microsoft.PowerShell.M... Waits for the pr...
Get-Counter                       Cmdlet    Microsoft.PowerShell.D... Gets performance...
Invoke-WSManAction                Cmdlet    Microsoft.WSMan.Manage... Invokes an actio...
Remove-WSManInstance              Cmdlet    Microsoft.WSMan.Manage... Deletes a manage...
Get-WSManInstance                 Cmdlet    Microsoft.WSMan.Manage... Displays managem...
New-WSManInstance                 Cmdlet    Microsoft.WSMan.Manage... Creates a new in...
Set-WSManInstance                 Cmdlet    Microsoft.WSMan.Manage... Modifies the man...
about_Arithmetic_Operators        HelpFile                            Describes the op...
```

```
about_Arrays                      HelpFile                           Describes arrays...
about_Debuggers                   HelpFile                           Describes the Wi...
about_Execution_Policies          HelpFile                           Describes the Wi...
about_ForEach-Parallel            HelpFile                           Describes the Fo...
about_Foreach                     HelpFile                           Describes a lang...
about_Functions                   HelpFile                           Describes how to...
about_Language_Keywords           HelpFile                           Describes the ke...
about_Methods                     HelpFile                           Describes how to...
about_Objects                     HelpFile                           Provides essenti...
about_Parallel                    HelpFile                           Describes the Pa...
about_Pipelines                   HelpFile                           Combining comman...
about_Preference_Variables        HelpFile                           Variables that c...
about_Remote                      HelpFile                           Describes how to...
about_Remote_Output               HelpFile                           Describes how to...
about_Sequence                    HelpFile                           Describes the Se...
about_Session_Configuration_Files HelpFile                           Describes sessio...
about_Variables                   HelpFile                           Describes how va...
about_Windows_PowerShell_5.0      HelpFile                           Describes new fe...
about_WQL                         HelpFile                           Describes WMI Qu...
about_WS-Management_Cmdlets       HelpFile                           Provides an over...
about_ForEach-Parallel            HelpFile                           Describes the Fo...
about_Parallel                    HelpFile                           Describes the Pa...
about_Sequence                    HelpFile                           Describes the Se...

PS C:\>
```

Using Help to search for “process” returned 10 results and using it to search for processes returned 68 results. If only one result is found, the help topic itself will be displayed instead of a list of commands.

```
PS C:\> get-help *hotfix*


NAME
    Get-HotFix

SYNOPSIS
    Gets the hotfixes that have been applied to the local and remote computers.


SYNTAX
    Get-HotFix [-ComputerName <String[]>] [-Credential <PSCredential>] [-Description
    <String[]>] [<CommonParameters>]

    Get-HotFix [[-Id] <String[]>] [-ComputerName <String[]>] [-Credential
    <PSCredential>] [<CommonParameters>]
```

```
DESCRIPTION
    The Get-Hotfix cmdlet gets hotfixes (also called updates) that have been installed
    on either the local computer (or on specified remote computers) by Windows Update,
    Microsoft Update, or Windows Server Update Services; the cmdlet also gets hotfixes
    or updates that have been installed manually by users.


RELATED LINKS
    Online Version: http://go.microsoft.com/fwlink/?LinkId=821586
    Win32_QuickFixEngineering http://go.microsoft.com/fwlink/?LinkID=145071
    Get-ComputerRestorePoint
    Add-Content

REMARKS
    To see the examples, type: "get-help Get-HotFix -examples".
    For more information, type: "get-help Get-HotFix -detailed".
    For technical information, type: "get-help Get-HotFix -full".
    For online help, type: "get-help Get-HotFix -online"

PS C:\>
```

Now to debunk the myth that Help in PowerShell can only find commands that have help topics.

```
PS C:\> help *more*


NAME
    more

SYNTAX
    more [[-paths] <string[]>]


ALIASES
    None


REMARKS
    None

PS C:\>
```

Notice in the previous example that more doesn't have a help topic, yet the Help system in PowerShell was able to find it. It only found one match and returned the basic syntax information that you'll see when a command doesn't have a help topic.

PowerShell contains numerous conceptual (About) help topics. The following command can be used to return a list of all About help topics on your system.

```
PS C:\> help About_*
```

Limiting the results to one single About help topic displays the actual help topic instead of returning a list.

```
PS C:\> help about_Updatable_Help
```

The help system in PowerShell has to be updated in order for the About help topics to be present. If for some reason the initial update of the help system failed on your Windows 10 lab environment computer, they will not be available until the Update-Help cmdlet has been run successfully which is covered later in this chapter.

Get-Command

Get-Command is designed to help you locate commands. Running Get-Command without any parameters returns a list of all the commands on your system. The following example demonstrates using the Get-Command cmdlet to determine what commands exist for working with processes:

```
PS C:\> Get-Command -Noun Process


CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Debug-Process                                      3.1.0.0    Microsof...
Cmdlet          Get-Process                                        3.1.0.0    Microsof...
Cmdlet          Start-Process                                      3.1.0.0    Microsof...
Cmdlet          Stop-Process                                       3.1.0.0    Microsof...
Cmdlet          Wait-Process                                       3.1.0.0    Microsof...

PS C:\>
```

Notice in the previous example where Get-Command was run, that the Noun parameter is used and Process is specified as the value for the noun parameter. What if you didn't know how to use the Get-Command cmdlet? You could simply use Get-Help to display the help topic for Get-Command.

The problem with the help topics in PowerShell is they're written by humans which means they're not perfect. According to the help topic for Get-Command, the Name, Noun, and Verb parameters don't allow wildcards. Those are parameters I use all the time and I can assure you they all do indeed accept wildcards, as shown in the following example where wildcards are used with the Name parameter.

```
PS C:\> Get-Command -Name *service*


CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Get-NetFirewallServiceFilter                       2.0.0.0    NetSecurity
Function        Set-NetFirewallServiceFilter                       2.0.0.0    NetSecurity
Cmdlet          Get-Service                                        3.1.0.0    Microsof...
Cmdlet          New-Service                                        3.1.0.0    Microsof...
Cmdlet          New-WebServiceProxy                                3.1.0.0    Microsof...
Cmdlet          Restart-Service                                    3.1.0.0    Microsof...
Cmdlet          Resume-Service                                     3.1.0.0    Microsof...
Cmdlet          Set-Service                                        3.1.0.0    Microsof...
Cmdlet          Start-Service                                      3.1.0.0    Microsof...
Cmdlet          Stop-Service                                       3.1.0.0    Microsof...
Cmdlet          Suspend-Service                                    3.1.0.0    Microsof...
Application     AgentService.exe                                   10.0.14... C:\Windo...
Application     SensorDataService.exe                              10.0.14... C:\Windo...
Application     services.exe                                       10.0.14... C:\Windo...
Application     services.msc                                       0.0.0.0    C:\Windo...
Application     TieringEngineService.exe                           10.0.14... C:\Windo...


PS C:\>
```

I'm not a big fan of using wildcards with the Name parameter of Get-Command since it also returns Windows executables that are not native PowerShell commands.

If you are going to use wildcard characters with the Name parameter, I recommend limiting the results with the CommandType parameter.

```
PS C:\> Get-Command -Name *service* -CommandType Cmdlet, Function, Alias
```

A better option is to use either the verb or noun parameter or both of them since only PowerShell commands have both verbs and nouns.

Found something wrong with a help topic such as the incorrect information described earlier in this chapter? The good news is the help topics for PowerShell have been open-sourced and now reside on GitHub. Pay it forward by not only fixing the incorrect information for yourself, but everyone else as well. Simply fork the PowerShell documentation repository on GitHub, update the help topic, and submit a pull request. Once the pull request is accepted, the corrected documentation will be available for everyone.

Updating Help

The local copy of the PowerShell help topics was previously updated the first time help on a command was requested. It's recommended to periodically update the help system because there can be updates to the help content from time to time. The Update-Help cmdlet is used to update the help topics. It requires Internet access by default and for you to be running PowerShell elevated as an administrator.

```
PS C:\> Update-Help


Update-Help : Failed to update Help for the module(s) 'BitsTransfer' with UI culture(s)
{en-US} : The value of the HelpInfoUri key in the module manifest must resolve to a
container or root URL on a website where the help files are stored. The HelpInfoUri
'https://technet.microsoft.com/en-us/library/dd819413.aspx' does not resolve to a
container.
At line:1 char:1
+ Update-Help
+
    + CategoryInfo          : InvalidOperation: (:) [Update-Help], Exception
    + FullyQualifiedErrorId : InvalidHelpInfoUri,Microsoft.PowerShell.Commands.UpdateHel
   pCommand

Update-Help : Failed to update Help for the module(s) 'NetworkControllerDiagnostics,
StorageReplica' with UI culture(s) {en-US} : Unable to retrieve the HelpInfo XML file
for UI culture en-US. Make sure the HelpInfoUri property in the module manifest is valid
or check your network connection and then try the command again.
At line:1 char:1
+ Update-Help
+
    + CategoryInfo          : ResourceUnavailable: (:) [Update-Help], Exception
    + FullyQualifiedErrorId : UnableToRetrieveHelpInfoXml,Microsoft.PowerShell.Commands.
   UpdateHelpCommand

PS C:\>
```

A couple of the modules returned errors which is not uncommon. If the machine didn't have Internet access, you could use the Save-Help cmdlet on another machine that does have Internet access to first save the updated help information to a file share on your network and then use the SourcePath parameter of Update-Help to specify this network location for the help topics.

Consider setting up a scheduled task or adding some logic to your profile script in PowerShell to periodically update the help content on your computer. Profile scripts will be discussed in an upcoming chapter.

Summary

In this chapter you've learned how to find commands with both Get-Help and Get-Command. You've learned how to use the help system to figure out how to use commands once you find them. You've also learned how to update the content of the help topics when updates are available.

My challenge to you is to learn a PowerShell command a day.

```
Get-Command | Get-Random | Get-Help -Full
```

Review

1. Is the DisplayName parameter of Get-Service positional?
2. How many parameter sets does the Get-Process cmdlet have?
3. What PowerShell commands exists for working with event logs?
4. What is the PowerShell command for returning a list of PowerShell processes running on your computer?
5. How do you update the PowerShell help content that's stored on your computer?

Recommended Reading

For those who want to know more information about the topics covered in this chapter, I recommend reading the following PowerShell help topics.

- Get-Help
- Get-Command
- Update-Help
- Save-Help
- about_Updatable_Help
- about_Command_Syntax

In the next chapter, you'll learn about the Get-Member cmdlet as well as objects, properties, and methods.

Chapter 3 - Discovering Objects, Properties, and Methods

My first introduction to computers was a Commodore 64, but my first modern computer was a 286 12Mhz IBM clone with 1 megabyte of memory, 40 megabyte hard drive, and one 5 1/4 inch floppy disk drive with a CGA monitor running Microsoft DOS 3.3.

Many IT Pros, like myself, are no stranger to the command line, but when the subject of objects, properties, and methods comes up, they get the deer in the headlights look and say, “I’m not a developer.” Guess what? You don’t have to be a developer to be extremely successful with PowerShell. Don’t get bogged down in the terminology. Read it and take it for what it’s worth. Not everything is going to make sense initially, but after a little hands on experience with PowerShell you’ll start to have those light bulb moments. Aha! So that’s what the book was talking about.

Be sure to follow along on your Windows 10 lab environment computer to gain some of that hands on experience with PowerShell.

Requirements

The Active Directory PowerShell module which installs as part of the RSAT (Remote Server Administration Tools) for Windows 10 is required by some of the examples shown in this chapter. They will also be used in subsequent chapters. When downloading the RSAT, choose the appropriate architecture (x64 if you’re running a 64bit version of Windows 10 or x86 if you’re running a 32bit version).

Get-Member

Get-Member helps you discover what objects, properties, and methods are available for commands. Any command that produces object based output can be piped to Get-Member. A property is a characteristic about an item. Your drivers license has a property called eye color and the most common values for that property are blue and brown. A method is an action that can be taken on an item. In staying with the drivers license example, one of the methods is “Revoke” because the department of motor vehicles can revoke your drivers license.

Properties

In the following example, I’ll retrieve information about the Windows Time service running on my Windows 10 lab environment computer.

```
PS C:\> Get-Service -Name w32time


Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time

PS C:\>
```

Status, Name, and DisplayName are examples of properties as shown in the previous set of results. The value for the Status property is Running, the value for the Name property is w32time, and the value for DisplayName is Windows Time.

Now I'll pipe that same command to Get-Member:

```
PS C:\> Get-Service -Name w32time | Get-Member


   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
RequiredServices          AliasProperty RequiredServices = ServicesDependedOn
Disposed                  Event         System.EventHandler Disposed(System.Object, Sy...
Close                     Method        void Close()
Continue                  Method        void Continue()
CreateObjRef              Method        System.Runtime.Remoting.ObjRef CreateObjRef(ty...
Dispose                   Method        void Dispose(), void IDisposable.Dispose()
Equals                    Method        bool Equals(System.Object obj)
ExecuteCommand            Method        void ExecuteCommand(int command)
GetHashCode               Method        int GetHashCode()
GetLifetimeService        Method        System.Object GetLifetimeService()
GetType                   Method        type GetType()
InitializeLifetimeService Method        System.Object InitializeLifetimeService()
Pause                     Method        void Pause()
Refresh                   Method        void Refresh()
Start                     Method        void Start(), void Start(string[] args)
Stop                      Method        void Stop()
WaitForStatus             Method        void WaitForStatus(System.ServiceProcess.Servi...
CanPauseAndContinue       Property      bool CanPauseAndContinue {get;}
CanShutdown               Property      bool CanShutdown {get;}
CanStop                   Property      bool CanStop {get;}
Container                 Property      System.ComponentModel.IContainer Container {get;}
DependentServices         Property      System.ServiceProcess.ServiceController[] Depe...
DisplayName               Property      string DisplayName {get;set;}
MachineName               Property      string MachineName {get;set;}
```

```
ServiceHandle          Property     System.Runtime.InteropServices.SafeHandle Serv...
ServiceName            Property     string ServiceName {get;set;}
ServicesDependedOn     Property     System.ServiceProcess.ServiceController[] Serv...
ServiceType            Property     System.ServiceProcess.ServiceType ServiceType ...
Site                   Property     System.ComponentModel.ISite Site {get;set;}
StartType              Property     System.ServiceProcess.ServiceStartMode StartTy...
Status                 Property     System.ServiceProcess.ServiceControllerStatus ...
ToString               ScriptMethod System.Object ToString();

PS C:\>
```

The first line of the results in the previous example contains one piece of very important information. TypeName tells you what type of object was returned. In this example, a System.ServiceProcess.ServiceController object was returned. This is often abbreviated as the portion of the TypeName just after the last period or ServiceController in this example.

Once you know what type of object a command produces, you'll be able to use this information to find commands which accept that type of object as input.

```
PS C:\> Get-Command -ParameterType ServiceController


CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Get-Service                                        3.1.0.0    Microsof...
Cmdlet          Restart-Service                                    3.1.0.0    Microsof...
Cmdlet          Resume-Service                                     3.1.0.0    Microsof...
Cmdlet          Set-Service                                        3.1.0.0    Microsof...
Cmdlet          Start-Service                                      3.1.0.0    Microsof...
Cmdlet          Stop-Service                                       3.1.0.0    Microsof...
Cmdlet          Suspend-Service                                    3.1.0.0    Microsof...

PS C:\>
```

All of those commands have a parameter which accepts a ServiceController object type by either pipeline or parameter input, or by both pipeline and parameter input.

Notice that there are more properties than are displayed by default. Although these additional properties aren't displayed by default, they can be selected from the pipeline by piping the command to the Select-Object cmdlet and using the Property parameter. The following example selects all of the properties by piping the results of Get-Service to Select-Object and specifying the * wildcard character as the value for the Property parameter.

```
PS C:\> Get-Service -Name w32time | Select-Object -Property *
```

```
Name                : w32time
RequiredServices    : {}
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
DisplayName         : Windows Time
DependentServices   : {}
MachineName         : .
ServiceName         : w32time
ServicesDependedOn  : {}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Manual
Site                :
Container           :

PS C:\>
```

Specific properties can also be selected by specifying them individually via a comma separated list for the value of the Property parameter.

```
PS C:\> Get-Service -Name w32time | Select-Object -Property Status, Name,
DisplayName, ServiceType
```

```
 Status Name    DisplayName        ServiceType
 ------ ----    -----------        -----------
Running w32time Windows Time Win32ShareProcess

PS C:\>
```

By default, four properties are returned in a table and five or more are returned in a list. Some commands use custom formatting to override how many properties are displayed by default in a table. There are several Format-* cmdlets which can be used to manually override these defaults. The most common ones are Format-Table and Format-List, both of which will be covered in an upcoming chapter.

Wildcard characters can be used when specifying the property names with Select-Object.

```
PS C:\> Get-Service -Name w32time | Select-Object -Property Status, DisplayName, Can*
```

```
Status              : Running
DisplayName         : Windows Time
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True

PS C:\>
```

In the previous example, Can* was used as one of the values for the property parameter which returned all of the properties that start with Can. These include CanPauseAndContinue, CanShutdown, and CanStop.

Methods

Methods are an action that can be taken. Use the MemberType parameter to narrow down the results of Get-Member to only show the methods for Get-Service.

```
PS C:\> Get-Service -Name w32time | Get-Member -MemberType Method
```

```
   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType Definition
----                      ---------- ----------
Close                     Method     void Close()
Continue                  Method     void Continue()
CreateObjRef              Method     System.Runtime.Remoting.ObjRef CreateObjRef(type ...
Dispose                   Method     void Dispose(), void IDisposable.Dispose()
Equals                    Method     bool Equals(System.Object obj)
ExecuteCommand            Method     void ExecuteCommand(int command)
GetHashCode               Method     int GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetType                   Method     type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
Pause                     Method     void Pause()
Refresh                   Method     void Refresh()
Start                     Method     void Start(), void Start(string[] args)
Stop                      Method     void Stop()
WaitForStatus             Method     void WaitForStatus(System.ServiceProcess.ServiceC...

PS C:\>
```

As you can see, there are numerous methods. The Stop method can be used to stop a Windows service.

```
PS C:\> (Get-Service -Name w32time).Stop()


PS C:\>
```

Now to verify the Windows time service has indeed been stopped.

```
PS C:\> Get-Service -Name w32time


Status   Name               DisplayName
------   ----               -----------
Stopped  w32time            Windows Time

PS C:\>
```

I rarely find myself using methods, but they're something you need to be aware of. There are times that you'll come across a Get-* command without a corresponding command to modify that item, but often times a method can be used to perform an action which modifies it. The Get-SqlAgentJob cmdlet that's part of the SqlServer PowerShell module which installs as part of SSMS (SQL Server Management Studio) 2016 is a good example of this. No corresponding Set cmdlet exists, but a method can be used to accomplish the same task.

Another reason to be aware of Methods is that a lot of beginners assume destructive changes can't be made with Get-* commands, but they indeed can cause a resume generating event if used inappropriately.

A better option is to use a cmdlet to perform the action if one exists. Go ahead and start the Windows time service, except this time use the cmdlet for starting services.

```
PS C:\> Get-Service -Name w32time | Start-Service -PassThru


Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time

PS C:\>
```

By default, Start-Service doesn't return any results just like the start method of Get-Service, but one of the benefits of using a cmdlet instead of a method is that many times the cmdlet offers additional functionality which is not available with a method. In the previous example, the PassThru parameter was specified which causes a cmdlet that doesn't normally produce output, to produce output.

Be careful with assumptions. We all know what happens when you assume things. I'll retrieve information about the PowerShell process running on my Windows 10 lab environment computer.

```
PS C:\> Get-Process -Name PowerShell


Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    922      48   107984     140552       2.84   9020   1 powershell


PS C:\>
```

Now I'll pipe that same command to Get-Member:

```
PS C:\> Get-Process -Name PowerShell | Get-Member


   TypeName: System.Diagnostics.Process

Name                       MemberType     Definition
----                       ----------     ----------
Handles                    AliasProperty  Handles = Handlecount
Name                       AliasProperty  Name = ProcessName
NPM                        AliasProperty  NPM = NonpagedSystemMemorySize64
PM                         AliasProperty  PM = PagedMemorySize64
SI                         AliasProperty  SI = SessionId
VM                         AliasProperty  VM = VirtualMemorySize64
WS                         AliasProperty  WS = WorkingSet64
Disposed                   Event          System.EventHandler Disposed(System.Object, ...
ErrorDataReceived          Event          System.Diagnostics.DataReceivedEventHandler ...
Exited                     Event          System.EventHandler Exited(System.Object, Sy...
OutputDataReceived         Event          System.Diagnostics.DataReceivedEventHandler ...
BeginErrorReadLine         Method         void BeginErrorReadLine()
BeginOutputReadLine        Method         void BeginOutputReadLine()
CancelErrorRead            Method         void CancelErrorRead()
CancelOutputRead           Method         void CancelOutputRead()
Close                      Method         void Close()
CloseMainWindow            Method         bool CloseMainWindow()
CreateObjRef               Method         System.Runtime.Remoting.ObjRef CreateObjRef(...
Dispose                    Method         void Dispose(), void IDisposable.Dispose()
Equals                     Method         bool Equals(System.Object obj)
GetHashCode                Method         int GetHashCode()
GetLifetimeService         Method         System.Object GetLifetimeService()
GetType                    Method         type GetType()
InitializeLifetimeService  Method         System.Object InitializeLifetimeService()
Kill                       Method         void Kill()
Refresh                    Method         void Refresh()
Start                      Method         bool Start()
ToString                   Method         string ToString()
```

```
WaitForExit                Method         bool WaitForExit(int milliseconds), void Wai...
WaitForInputIdle           Method         bool WaitForInputIdle(int milliseconds), boo...
__NounName                 NoteProperty   string __NounName=Process
BasePriority               Property       int BasePriority {get;}
Container                  Property       System.ComponentModel.IContainer Container {...
EnableRaisingEvents        Property       bool EnableRaisingEvents {get;set;}
ExitCode                   Property       int ExitCode {get;}
ExitTime                   Property       datetime ExitTime {get;}
Handle                     Property       System.IntPtr Handle {get;}
HandleCount                Property       int HandleCount {get;}
HasExited                  Property       bool HasExited {get;}
Id                         Property       int Id {get;}
MachineName                Property       string MachineName {get;}
MainModule                 Property       System.Diagnostics.ProcessModule MainModule ...
MainWindowHandle           Property       System.IntPtr MainWindowHandle {get;}
MainWindowTitle            Property       string MainWindowTitle {get;}
MaxWorkingSet              Property       System.IntPtr MaxWorkingSet {get;set;}
MinWorkingSet              Property       System.IntPtr MinWorkingSet {get;set;}
Modules                    Property       System.Diagnostics.ProcessModuleCollection M...
NonpagedSystemMemorySize   Property       int NonpagedSystemMemorySize {get;}
NonpagedSystemMemorySize64 Property       long NonpagedSystemMemorySize64 {get;}
PagedMemorySize            Property       int PagedMemorySize {get;}
PagedMemorySize64          Property       long PagedMemorySize64 {get;}
PagedSystemMemorySize      Property       int PagedSystemMemorySize {get;}
PagedSystemMemorySize64    Property       long PagedSystemMemorySize64 {get;}
PeakPagedMemorySize        Property       int PeakPagedMemorySize {get;}
PeakPagedMemorySize64      Property       long PeakPagedMemorySize64 {get;}
PeakVirtualMemorySize      Property       int PeakVirtualMemorySize {get;}
PeakVirtualMemorySize64    Property       long PeakVirtualMemorySize64 {get;}
PeakWorkingSet             Property       int PeakWorkingSet {get;}
PeakWorkingSet64           Property       long PeakWorkingSet64 {get;}
PriorityBoostEnabled       Property       bool PriorityBoostEnabled {get;set;}
PriorityClass              Property       System.Diagnostics.ProcessPriorityClass Prio...
PrivateMemorySize          Property       int PrivateMemorySize {get;}
PrivateMemorySize64        Property       long PrivateMemorySize64 {get;}
PrivilegedProcessorTime    Property       timespan PrivilegedProcessorTime {get;}
ProcessName                Property       string ProcessName {get;}
ProcessorAffinity          Property       System.IntPtr ProcessorAffinity {get;set;}
Responding                 Property       bool Responding {get;}
SafeHandle                 Property       Microsoft.Win32.SafeHandles.SafeProcessHandl...
SessionId                  Property       int SessionId {get;}
Site                       Property       System.ComponentModel.ISite Site {get;set;}
StandardError              Property       System.IO.StreamReader StandardError {get;}
StandardInput              Property       System.IO.StreamWriter StandardInput {get;}
StandardOutput             Property       System.IO.StreamReader StandardOutput {get;}
StartInfo                  Property       System.Diagnostics.ProcessStartInfo StartInf...
StartTime                  Property       datetime StartTime {get;}
```

```
SynchronizingObject        Property       System.ComponentModel.ISynchronizeInvoke Syn...
Threads                    Property       System.Diagnostics.ProcessThreadCollection T...
TotalProcessorTime         Property       timespan TotalProcessorTime {get;}
UserProcessorTime          Property       timespan UserProcessorTime {get;}
VirtualMemorySize          Property       int VirtualMemorySize {get;}
VirtualMemorySize64        Property       long VirtualMemorySize64 {get;}
WorkingSet                 Property       int WorkingSet {get;}
WorkingSet64               Property       long WorkingSet64 {get;}
PSConfiguration            PropertySet    PSConfiguration {Name, Id, PriorityClass, Fi...
PSResources                PropertySet    PSResources {Name, Id, Handlecount, WorkingS...
Company                    ScriptProperty System.Object Company {get=$this.Mainmodule....
CPU                        ScriptProperty System.Object CPU {get=$this.TotalProcessorT...
Description                ScriptProperty System.Object Description {get=$this.Mainmod...
FileVersion                ScriptProperty System.Object FileVersion {get=$this.Mainmod...
Path                       ScriptProperty System.Object Path {get=$this.Mainmodule.Fil...
Product                    ScriptProperty System.Object Product {get=$this.Mainmodule....
ProductVersion             ScriptProperty System.Object ProductVersion {get=$this.Main...


PS C:\>
```

Notice that in addition to there being more properties than are displayed by default, that a number of the default properties don't show up as being properties when viewing the results of piping the command to Get-Member. This is because many of the default results such as NPM(K), PM(K), WS(K), and CPU(s) are calculated properties. In order to determine the actual property names, the command must be piped to Get-Member.

If a command does not produce output, it can't be piped to Get-Member. Since Start-Service doesn't produce any output by default, it generates an error when you try to pipe it to Get-Member.

```
PS C:\> Start-Service -Name w32time | Get-Member


Get-Member : You must specify an object for the Get-Member cmdlet.
At line:1 char:31
+ Start-Service -Name w32time | Get-Member
+
    + CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperationException
    + FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Commands.GetMembe
   rCommand

PS C:\>
```

The PassThru parameter which was previously mentioned can be specified with the Start-Service cmdlet in order to make it produce output and then it can be piped to Get-Member without error.

```
PS C:\> Start-Service -Name w32time -PassThru | Get-Member


   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
RequiredServices          AliasProperty RequiredServices = ServicesDependedOn
Disposed                  Event         System.EventHandler Disposed(System.Object, Sy...
Close                     Method        void Close()
Continue                  Method        void Continue()
CreateObjRef              Method        System.Runtime.Remoting.ObjRef CreateObjRef(ty...
Dispose                   Method        void Dispose(), void IDisposable.Dispose()
Equals                    Method        bool Equals(System.Object obj)
ExecuteCommand            Method        void ExecuteCommand(int command)
GetHashCode               Method        int GetHashCode()
GetLifetimeService        Method        System.Object GetLifetimeService()
GetType                   Method        type GetType()
InitializeLifetimeService Method        System.Object InitializeLifetimeService()
Pause                     Method        void Pause()
Refresh                   Method        void Refresh()
Start                     Method        void Start(), void Start(string[] args)
Stop                      Method        void Stop()
WaitForStatus             Method        void WaitForStatus(System.ServiceProcess.Servi...
CanPauseAndContinue       Property      bool CanPauseAndContinue {get;}
CanShutdown               Property      bool CanShutdown {get;}
CanStop                   Property      bool CanStop {get;}
Container                 Property      System.ComponentModel.IContainer Container {get;}
DependentServices         Property      System.ServiceProcess.ServiceController[] Depe...
DisplayName               Property      string DisplayName {get;set;}
MachineName               Property      string MachineName {get;set;}
ServiceHandle             Property      System.Runtime.InteropServices.SafeHandle Serv...
ServiceName               Property      string ServiceName {get;set;}
ServicesDependedOn        Property      System.ServiceProcess.ServiceController[] Serv...
ServiceType               Property      System.ServiceProcess.ServiceType ServiceType ...
Site                      Property      System.ComponentModel.ISite Site {get;set;}
StartType                 Property      System.ServiceProcess.ServiceStartMode StartTy...
Status                    Property      System.ServiceProcess.ServiceControllerStatus ...
ToString                  ScriptMethod  System.Object ToString();

 PS C:\>
```

In order to be piped to Get-Member, a command must not only produce output, but it must also produce object based output.

```
PS C:\> Get-Service -Name w32time | Out-Host | Get-Member


Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time

Get-Member : You must specify an object for the Get-Member cmdlet.
At line:1 char:40
+ Get-Service -Name w32time | Out-Host | Get-Member
+
    + CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperationException
    + FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Commands.GetMembe
   rCommand

PS C:\>
```

Although Out-Host produces output, it doesn't produce object based output so it can't be piped to Get-Member.

Active Directory

The remote server administration tools which are listed in the requirements section of this chapter are required to complete this section. Also, as mentioned in the lab environment section of the introduction to this book, your Windows 10 lab environment computer must be a member of a lab environment (non-production) Active Directory domain.

Use Get-Command with the Module parameter to determine what commands were added as part of the ActiveDirectory PowerShell module when the remote server administration tools were installed.

```
PS C:\> Get-Command -Module ActiveDirectory


CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Add-ADCentralAccessPolicyMember                    1.0.0.0    ActiveDi...
Cmdlet          Add-ADComputerServiceAccount                       1.0.0.0    ActiveDi...
Cmdlet          Add-ADDomainControllerPasswordReplicationPolicy    1.0.0.0    ActiveDi...
Cmdlet          Add-ADFineGrainedPasswordPolicySubject             1.0.0.0    ActiveDi...
Cmdlet          Add-ADGroupMember                                  1.0.0.0    ActiveDi...
Cmdlet          Add-ADPrincipalGroupMembership                     1.0.0.0    ActiveDi...
Cmdlet          Add-ADResourcePropertyListMember                   1.0.0.0    ActiveDi...
Cmdlet          Clear-ADAccountExpiration                          1.0.0.0    ActiveDi...
Cmdlet          Clear-ADClaimTransformLink                         1.0.0.0    ActiveDi...
Cmdlet          Disable-ADAccount                                  1.0.0.0    ActiveDi...
Cmdlet          Disable-ADOptionalFeature                          1.0.0.0    ActiveDi...
Cmdlet          Enable-ADAccount                                   1.0.0.0    ActiveDi...
Cmdlet          Enable-ADOptionalFeature                           1.0.0.0    ActiveDi...
Cmdlet          Get-ADAccountAuthorizationGroup                    1.0.0.0    ActiveDi...
Cmdlet          Get-ADAccountResultantPasswordReplicationPolicy    1.0.0.0    ActiveDi...
Cmdlet          Get-ADAuthenticationPolicy                         1.0.0.0    ActiveDi...
Cmdlet          Get-ADAuthenticationPolicySilo                     1.0.0.0    ActiveDi...
Cmdlet          Get-ADCentralAccessPolicy                          1.0.0.0    ActiveDi...
Cmdlet          Get-ADCentralAccessRule                            1.0.0.0    ActiveDi...
Cmdlet          Get-ADClaimTransformPolicy                         1.0.0.0    ActiveDi...
Cmdlet          Get-ADClaimType                                    1.0.0.0    ActiveDi...
Cmdlet          Get-ADComputer                                     1.0.0.0    ActiveDi...
Cmdlet          Get-ADComputerServiceAccount                       1.0.0.0    ActiveDi...
Cmdlet          Get-ADDCCloningExcludedApplicationList             1.0.0.0    ActiveDi...
Cmdlet          Get-ADDefaultDomainPasswordPolicy                  1.0.0.0    ActiveDi...
Cmdlet          Get-ADDomain                                       1.0.0.0    ActiveDi...
Cmdlet          Get-ADDomainController                             1.0.0.0    ActiveDi...
Cmdlet          Get-ADDomainControllerPasswordReplicationPolicy    1.0.0.0    ActiveDi...
Cmdlet          Get-ADDomainControllerPasswordReplicationPolicy... 1.0.0.0    ActiveDi...
Cmdlet          Get-ADFineGrainedPasswordPolicy                    1.0.0.0    ActiveDi...
Cmdlet          Get-ADFineGrainedPasswordPolicySubject             1.0.0.0    ActiveDi...
Cmdlet          Get-ADForest                                       1.0.0.0    ActiveDi...
Cmdlet          Get-ADGroup                                        1.0.0.0    ActiveDi...
Cmdlet          Get-ADGroupMember                                  1.0.0.0    ActiveDi...
Cmdlet          Get-ADObject                                       1.0.0.0    ActiveDi...
Cmdlet          Get-ADOptionalFeature                              1.0.0.0    ActiveDi...
Cmdlet          Get-ADOrganizationalUnit                           1.0.0.0    ActiveDi...
Cmdlet          Get-ADPrincipalGroupMembership                     1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationAttributeMetadata                 1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationConnection                        1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationFailure                           1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationPartnerMetadata                   1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationQueueOperation                    1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationSite                              1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationSiteLink                          1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationSiteLinkBridge                    1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationSubnet                            1.0.0.0    ActiveDi...
Cmdlet          Get-ADReplicationUpToDatenessVectorTable           1.0.0.0    ActiveDi...
Cmdlet          Get-ADResourceProperty                             1.0.0.0    ActiveDi...
Cmdlet          Get-ADResourcePropertyList                         1.0.0.0    ActiveDi...
Cmdlet          Get-ADResourcePropertyValueType                    1.0.0.0    ActiveDi...
Cmdlet          Get-ADRootDSE                                      1.0.0.0    ActiveDi...
Cmdlet          Get-ADServiceAccount                               1.0.0.0    ActiveDi...
Cmdlet          Get-ADTrust                                        1.0.0.0    ActiveDi...
Cmdlet          Get-ADUser                                         1.0.0.0    ActiveDi...
Cmdlet          Get-ADUserResultantPasswordPolicy                  1.0.0.0    ActiveDi...
Cmdlet          Grant-ADAuthenticationPolicySiloAccess             1.0.0.0    ActiveDi...
Cmdlet          Install-ADServiceAccount                           1.0.0.0    ActiveDi...
Cmdlet          Move-ADDirectoryServer                             1.0.0.0    ActiveDi...
Cmdlet          Move-ADDirectoryServerOperationMasterRole          1.0.0.0    ActiveDi...
Cmdlet          Move-ADObject                                      1.0.0.0    ActiveDi...
Cmdlet          New-ADAuthenticationPolicy                         1.0.0.0    ActiveDi...
Cmdlet          New-ADAuthenticationPolicySilo                     1.0.0.0    ActiveDi...
Cmdlet          New-ADCentralAccessPolicy                          1.0.0.0    ActiveDi...
Cmdlet          New-ADCentralAccessRule                            1.0.0.0    ActiveDi...
Cmdlet          New-ADClaimTransformPolicy                         1.0.0.0    ActiveDi...
Cmdlet          New-ADClaimType                                    1.0.0.0    ActiveDi...
Cmdlet          New-ADComputer                                     1.0.0.0    ActiveDi...
Cmdlet          New-ADDCCloneConfigFile                            1.0.0.0    ActiveDi...
Cmdlet          New-ADFineGrainedPasswordPolicy                    1.0.0.0    ActiveDi...
Cmdlet          New-ADGroup                                        1.0.0.0    ActiveDi...
Cmdlet          New-ADObject                                       1.0.0.0    ActiveDi...
Cmdlet          New-ADOrganizationalUnit                           1.0.0.0    ActiveDi...
Cmdlet          New-ADReplicationSite                              1.0.0.0    ActiveDi...
Cmdlet          New-ADReplicationSiteLink                          1.0.0.0    ActiveDi...
Cmdlet          New-ADReplicationSiteLinkBridge                    1.0.0.0    ActiveDi...
Cmdlet          New-ADReplicationSubnet                            1.0.0.0    ActiveDi...
Cmdlet          New-ADResourceProperty                             1.0.0.0    ActiveDi...
Cmdlet          New-ADResourcePropertyList                         1.0.0.0    ActiveDi...
Cmdlet          New-ADServiceAccount                               1.0.0.0    ActiveDi...
Cmdlet          New-ADUser                                         1.0.0.0    ActiveDi...
Cmdlet          Remove-ADAuthenticationPolicy                      1.0.0.0    ActiveDi...
Cmdlet          Remove-ADAuthenticationPolicySilo                  1.0.0.0    ActiveDi...
Cmdlet          Remove-ADCentralAccessPolicy                       1.0.0.0    ActiveDi...
Cmdlet          Remove-ADCentralAccessPolicyMember                 1.0.0.0    ActiveDi...
Cmdlet          Remove-ADCentralAccessRule                         1.0.0.0    ActiveDi...
Cmdlet          Remove-ADClaimTransformPolicy                      1.0.0.0    ActiveDi...
Cmdlet          Remove-ADClaimType                                 1.0.0.0    ActiveDi...
Cmdlet          Remove-ADComputer                                  1.0.0.0    ActiveDi...
Cmdlet          Remove-ADComputerServiceAccount                    1.0.0.0    ActiveDi...
Cmdlet          Remove-ADDomainControllerPasswordReplicationPolicy 1.0.0.0    ActiveDi...
Cmdlet          Remove-ADFineGrainedPasswordPolicy                 1.0.0.0    ActiveDi...
Cmdlet          Remove-ADFineGrainedPasswordPolicySubject          1.0.0.0    ActiveDi...
Cmdlet          Remove-ADGroup                                     1.0.0.0    ActiveDi...
Cmdlet          Remove-ADGroupMember                               1.0.0.0    ActiveDi...
Cmdlet          Remove-ADObject                                    1.0.0.0    ActiveDi...
Cmdlet          Remove-ADOrganizationalUnit                        1.0.0.0    ActiveDi...
Cmdlet          Remove-ADPrincipalGroupMembership                  1.0.0.0    ActiveDi...
Cmdlet          Remove-ADReplicationSite                           1.0.0.0    ActiveDi...
Cmdlet          Remove-ADReplicationSiteLink                       1.0.0.0    ActiveDi...
Cmdlet          Remove-ADReplicationSiteLinkBridge                 1.0.0.0    ActiveDi...
Cmdlet          Remove-ADReplicationSubnet                         1.0.0.0    ActiveDi...
Cmdlet          Remove-ADResourceProperty                          1.0.0.0    ActiveDi...
Cmdlet          Remove-ADResourcePropertyList                      1.0.0.0    ActiveDi...
Cmdlet          Remove-ADResourcePropertyListMember                1.0.0.0    ActiveDi...
Cmdlet          Remove-ADServiceAccount                            1.0.0.0    ActiveDi...
Cmdlet          Remove-ADUser                                      1.0.0.0    ActiveDi...
Cmdlet          Rename-ADObject                                    1.0.0.0    ActiveDi...
Cmdlet          Reset-ADServiceAccountPassword                     1.0.0.0    ActiveDi...
Cmdlet          Restore-ADObject                                   1.0.0.0    ActiveDi...
Cmdlet          Revoke-ADAuthenticationPolicySiloAccess            1.0.0.0    ActiveDi...
Cmdlet          Search-ADAccount                                   1.0.0.0    ActiveDi...
Cmdlet          Set-ADAccountAuthenticationPolicySilo              1.0.0.0    ActiveDi...
Cmdlet          Set-ADAccountControl                               1.0.0.0    ActiveDi...
Cmdlet          Set-ADAccountExpiration                            1.0.0.0    ActiveDi...
Cmdlet          Set-ADAccountPassword                              1.0.0.0    ActiveDi...
Cmdlet          Set-ADAuthenticationPolicy                         1.0.0.0    ActiveDi...
Cmdlet          Set-ADAuthenticationPolicySilo                     1.0.0.0    ActiveDi...
Cmdlet          Set-ADCentralAccessPolicy                          1.0.0.0    ActiveDi...
Cmdlet          Set-ADCentralAccessRule                            1.0.0.0    ActiveDi...
Cmdlet          Set-ADClaimTransformLink                           1.0.0.0    ActiveDi...
Cmdlet          Set-ADClaimTransformPolicy                         1.0.0.0    ActiveDi...
Cmdlet          Set-ADClaimType                                    1.0.0.0    ActiveDi...
Cmdlet          Set-ADComputer                                     1.0.0.0    ActiveDi...
Cmdlet          Set-ADDefaultDomainPasswordPolicy                  1.0.0.0    ActiveDi...
Cmdlet          Set-ADDomain                                       1.0.0.0    ActiveDi...
Cmdlet          Set-ADDomainMode                                   1.0.0.0    ActiveDi...
Cmdlet          Set-ADFineGrainedPasswordPolicy                    1.0.0.0    ActiveDi...
Cmdlet          Set-ADForest                                       1.0.0.0    ActiveDi...
Cmdlet          Set-ADForestMode                                   1.0.0.0    ActiveDi...
Cmdlet          Set-ADGroup                                        1.0.0.0    ActiveDi...
Cmdlet          Set-ADObject                                       1.0.0.0    ActiveDi...
Cmdlet          Set-ADOrganizationalUnit                           1.0.0.0    ActiveDi...
Cmdlet          Set-ADReplicationConnection                        1.0.0.0    ActiveDi...
Cmdlet          Set-ADReplicationSite                              1.0.0.0    ActiveDi...
Cmdlet          Set-ADReplicationSiteLink                          1.0.0.0    ActiveDi...
Cmdlet          Set-ADReplicationSiteLinkBridge                    1.0.0.0    ActiveDi...
Cmdlet          Set-ADReplicationSubnet                            1.0.0.0    ActiveDi...
Cmdlet          Set-ADResourceProperty                             1.0.0.0    ActiveDi...
Cmdlet          Set-ADResourcePropertyList                         1.0.0.0    ActiveDi...
Cmdlet          Set-ADServiceAccount                               1.0.0.0    ActiveDi...
Cmdlet          Set-ADUser                                         1.0.0.0    ActiveDi...
Cmdlet          Show-ADAuthenticationPolicyExpression              1.0.0.0    ActiveDi...
Cmdlet          Sync-ADObject                                      1.0.0.0    ActiveDi...
Cmdlet          Test-ADServiceAccount                              1.0.0.0    ActiveDi...
Cmdlet          Uninstall-ADServiceAccount                         1.0.0.0    ActiveDi...
Cmdlet          Unlock-ADAccount                                   1.0.0.0    ActiveDi...

PS C:\>
```

A total of 147 commands were added as part of the ActiveDirectory PowerShell module.

Some commands, such as many of the ones that are part of the ActiveDirectory PowerShell module, only return a portion of the available properties by default.

```
PS C:\> Get-ADUser -Identity mike | Get-Member


   TypeName: Microsoft.ActiveDirectory.Management.ADUser

Name              MemberType            Definition
----              ----------            ----------
Contains          Method                bool Contains(string propertyName)
Equals            Method                bool Equals(System.Object obj)
GetEnumerator     Method                System.Collections.IDictionaryEnumerator GetEn...
GetHashCode       Method                int GetHashCode()
GetType           Method                type GetType()
ToString          Method                string ToString()
Item              ParameterizedProperty Microsoft.ActiveDirectory.Management.ADPropert...
DistinguishedName Property              System.String DistinguishedName {get;set;}
Enabled           Property              System.Boolean Enabled {get;set;}
GivenName         Property              System.String GivenName {get;set;}
Name              Property              System.String Name {get;}
ObjectClass       Property              System.String ObjectClass {get;set;}
ObjectGUID        Property              System.Nullable`1[[System.Guid, mscorlib, Vers...
SamAccountName    Property              System.String SamAccountName {get;set;}
SID               Property              System.Security.Principal.SecurityIdentifier S...
Surname           Property              System.String Surname {get;set;}
UserPrincipalName Property              System.String UserPrincipalName {get;set;}

PS C:\>
```

Even if you're only vaguely familiar with Active Directory, you're probably aware that an Active Directory user account has more properties than are shown in the previous set of results.

The Get-ADUser cmdlet has a properties parameter which is used to specify which additional (non-default) properties you want to return. Specifying the * wildcard character returns all of them.

```
PS C:\> Get-ADUser -Identity mike -Properties * | Get-Member
```

```
   TypeName: Microsoft.ActiveDirectory.Management.ADUser

Name                              MemberType            Definition
----                              ----------            ----------
Contains                          Method                bool Contains(string proper...
Equals                            Method                bool Equals(System.Object obj)
GetEnumerator                     Method                System.Collections.IDiction...
GetHashCode                       Method                int GetHashCode()
GetType                           Method                type GetType()
ToString                          Method                string ToString()
Item                              ParameterizedProperty Microsoft.ActiveDirectory.M...
AccountExpirationDate             Property              System.DateTime AccountExpi...
accountExpires                    Property              System.Int64 accountExpires...
AccountLockoutTime                Property              System.DateTime AccountLock...
AccountNotDelegated               Property              System.Boolean AccountNotDe...
AllowReversiblePasswordEncryption Property              System.Boolean AllowReversi...
AuthenticationPolicy              Property              Microsoft.ActiveDirectory.M...
AuthenticationPolicySilo          Property              Microsoft.ActiveDirectory.M...
BadLogonCount                     Property              System.Int32 BadLogonCount ...
badPasswordTime                   Property              System.Int64 badPasswordTim...
badPwdCount                       Property              System.Int32 badPwdCount {g...
CannotChangePassword              Property              System.Boolean CannotChange...
CanonicalName                     Property              System.String CanonicalName...
Certificates                      Property              Microsoft.ActiveDirectory.M...
City                              Property              System.String City {get;set;}
CN                                Property              System.String CN {get;}
codePage                          Property              System.Int32 codePage {get;...
Company                           Property              System.String Company {get;...
CompoundIdentitySupported         Property              Microsoft.ActiveDirectory.M...
Country                           Property              System.String Country {get;...
countryCode                       Property              System.Int32 countryCode {g...
Created                           Property              System.DateTime Created {get;}
createTimeStamp                   Property              System.DateTime createTimeS...
Deleted                           Property              System.Boolean Deleted {get;}
Department                        Property              System.String Department {g...
Description                       Property              System.String Description {...
DisplayName                       Property              System.String DisplayName {...
DistinguishedName                 Property              System.String Distinguished...
Division                          Property              System.String Division {get...
DoesNotRequirePreAuth             Property              System.Boolean DoesNotRequi...
dSCorePropagationData             Property              Microsoft.ActiveDirectory.M...
EmailAddress                      Property              System.String EmailAddress ...
EmployeeID                        Property              System.String EmployeeID {g...
EmployeeNumber                    Property              System.String EmployeeNumbe...
Enabled                           Property              System.Boolean Enabled {get...
Fax                               Property              System.String Fax {get;set;}
GivenName                         Property              System.String GivenName {ge...
```

```
HomeDirectory                          Property          System.String HomeDirectory...
HomedirRequired                        Property          System.Boolean HomedirRequi...
HomeDrive                              Property          System.String HomeDrive {ge...
HomePage                               Property          System.String HomePage {get...
HomePhone                              Property          System.String HomePhone {ge...
Initials                               Property          System.String Initials {get...
instanceType                           Property          System.Int32 instanceType {...
isDeleted                              Property          System.Boolean isDeleted {g...
KerberosEncryptionType                 Property          Microsoft.ActiveDirectory.M...
LastBadPasswordAttempt                 Property          System.DateTime LastBadPass...
LastKnownParent                        Property          System.String LastKnownPare...
lastLogoff                             Property          System.Int64 lastLogoff {ge...
lastLogon                              Property          System.Int64 lastLogon {get...
LastLogonDate                          Property          System.DateTime LastLogonDa...
lastLogonTimestamp                     Property          System.Int64 lastLogonTimes...
LockedOut                              Property          System.Boolean LockedOut {g...
logonCount                             Property          System.Int32 logonCount {ge...
LogonWorkstations                      Property          System.String LogonWorkstat...
Manager                                Property          System.String Manager {get;...
MemberOf                               Property          Microsoft.ActiveDirectory.M...
MNSLogonAccount                        Property          System.Boolean MNSLogonAcco...
MobilePhone                            Property          System.String MobilePhone {...
Modified                               Property          System.DateTime Modified {g...
modifyTimeStamp                        Property          System.DateTime modifyTimeS...
msDS-User-Account-Control-Computed     Property          System.Int32 msDS-User-Acco...
Name                                   Property          System.String Name {get;}
nTSecurityDescriptor                   Property          System.DirectoryServices.Ac...
ObjectCategory                         Property          System.String ObjectCategor...
ObjectClass                            Property          System.String ObjectClass {...
ObjectGUID                             Property          System.Nullable`1[[System.G...
objectSid                              Property          System.Security.Principal.S...
Office                                 Property          System.String Office {get;s...
OfficePhone                            Property          System.String OfficePhone {...
Organization                           Property          System.String Organization ...
OtherName                              Property          System.String OtherName {ge...
PasswordExpired                        Property          System.Boolean PasswordExpi...
PasswordLastSet                        Property          System.DateTime PasswordLas...
PasswordNeverExpires                   Property          System.Boolean PasswordNeve...
PasswordNotRequired                    Property          System.Boolean PasswordNotR...
POBox                                  Property          System.String POBox {get;set;}
PostalCode                             Property          System.String PostalCode {g...
PrimaryGroup                           Property          System.String PrimaryGroup ...
primaryGroupID                         Property          System.Int32 primaryGroupID...
PrincipalsAllowedToDelegateToAccount Property          Microsoft.ActiveDirectory.M...
ProfilePath                            Property          System.String ProfilePath {...
ProtectedFromAccidentalDeletion        Property          System.Boolean ProtectedFro...
pwdAnswer                              Property          System.String pwdAnswer {ge...
```

```
pwdLastSet                          Property             System.Int64 pwdLastSet {ge...
pwdQuestion                         Property             System.String pwdQuestion {...
SamAccountName                      Property             System.String SamAccountNam...
sAMAccountType                      Property             System.Int32 sAMAccountType...
ScriptPath                          Property             System.String ScriptPath {g...
sDRightsEffective                   Property             System.Int32 sDRightsEffect...
ServicePrincipalNames               Property             Microsoft.ActiveDirectory.M...
SID                                 Property             System.Security.Principal.S...
SIDHistory                          Property             Microsoft.ActiveDirectory.M...
SmartcardLogonRequired              Property             System.Boolean SmartcardLog...
sn                                  Property             System.String sn {get;set;}
State                               Property             System.String State {get;set;}
StreetAddress                       Property             System.String StreetAddress...
Surname                             Property             System.String Surname {get;...
Title                               Property             System.String Title {get;set;}
TrustedForDelegation                Property             System.Boolean TrustedForDe...
TrustedToAuthForDelegation          Property             System.Boolean TrustedToAut...
UseDESKeyOnly                       Property             System.Boolean UseDESKeyOnl...
userAccountControl                  Property             System.Int32 userAccountCon...
userCertificate                     Property             Microsoft.ActiveDirectory.M...
UserPrincipalName                   Property             System.String UserPrincipal...
uSNChanged                          Property             System.Int64 uSNChanged {get;}
uSNCreated                          Property             System.Int64 uSNCreated {get;}
whenChanged                         Property             System.DateTime whenChanged...
whenCreated                         Property             System.DateTime whenCreated...

PS C:\>
```

Now that looks more like it.

Can you think of a reason why the properties of an Active Directory user account would be so limited by default? Imagine if you returned every property for every user account in your production Active Directory environment. Think of the performance degradation that you could cause not only to the domain controllers themselves, but also to your network and it's highly doubtful that you'll actually need every property anyway. Returning all of the properties for a single user account as shown in the previous example is perfectly acceptable when you're trying to figure what properties exist.

It's not uncommon to run a command numerous times when prototyping it. If you're going to perform some huge query from something such as Active Directory, query it once and store the results in a variable and then work with the contents of the variable instead of constantly performing some expensive query over and over again.

```
PS C:\> $Users = Get-ADUser -Identity mike -Properties *
```

Use the contents of the Users variable instead of running the previous command numerous times. Keep in mind that the contents of the variable won't be updated if you make some change to that particular user in Active Directory.

I've heard Ed Wilson, the Scripting Guy, use the analogy "if you're going to eat an elephant only eat it once" to describe this technique.

You could pipe the Users variable to Get-Member to determine what the properties are.

```
PS C:\> $Users | Get-Member
```

Then select the individual properties by piping the Users variable to Select-Object, all without ever having to query Active Directory more than one time.

```
PS C:\> $Users | Select-Object -Property Name, LastLogonDate, LastBadPasswordAttempt
```

If you are going to query Active Directory more than one time, specify any of the non-default properties individually via the Properties parameter once you've determined what they are.

```
PS C:\> Get-ADUser -Identity mike -Properties LastLogonDate, LastBadPasswordAttempt


DistinguishedName      : CN=Mike F. Robbins,OU=Sales,DC=mikefrobbins,DC=com
Enabled                : True
GivenName              : Mike
LastBadPasswordAttempt : 2/4/2017 10:46:15 AM
LastLogonDate          : 2/18/2017 12:45:14 AM
Name                   : Mike F. Robbins
ObjectClass            : user
ObjectGUID             : a82a8c58-1332-4a57-a6e2-68e0c750ea56
SamAccountName         : mike
SID                    : S-1-5-21-2989741381-570885089-3319121794-1108
Surname                : Robbins
UserPrincipalName      : miker@mikefrobbins.com

PS C:\>
```

Did you notice anything different about the names of the commands that are part of the ActiveDirectory PowerShell module? The noun portion of the commands has an AD prefix. This is something that's common to see on most commands with the exception of the ones for administering Microsoft Exchange Server and that's because Exchange was the first product in which Microsoft added PowerShell support. This prefix is designed to help prevent naming conflicts.

Summary

In this chapter, you've learned how to determine what type of object a command produces, how to determine what properties and methods are available for a command, and how to work with commands that limit the properties that are returned by default.

Review

1. What type of object does the Get-Process cmdlet produce?
2. How do you determine what the available properties are for a command?
3. If a command exists for getting something but not for setting the same thing, what should you check for?
4. How can certain commands that don't produce output by default be made to produce output?
5. If you're going to be working with the results of a command that produces an enormous amount of output, what should you consider doing?

Recommended Reading

- Get-Member
- Viewing Object Structure (Get-Member)
- about_Objects
- about_Properties
- about_Methods
- No PowerShell Cmdlet to Start or Stop Something? Don't Forget to Check for Methods on the Get Cmdlets

Chapter 4 - One-Liners and the Pipeline

When I first started learning PowerShell, if I couldn't accomplish a task with a PowerShell one-liner, I went back to the GUI and used it. Over time, I built my skills up to writing scripts, functions, and modules. Don't allow yourself to become overwhelmed with some of the more advanced examples you may see on the Internet because no one is a natural expert with PowerShell. We were all beginners at one point in time.

One bit of advice that I'll offer to those of you who are still using the GUI for some of your administration, as I was when I first started learning PowerShell, is to install the management tools on your workstation or a jump server and manage your servers remotely. This way it won't matter if the server is running a GUI or the server core installation of the operating system. It's going to help prepare you for managing servers remotely with PowerShell.

As with previous chapters, be sure to follow along on your Windows 10 lab environment computer.

One-Liners

A PowerShell one-liner is one continuous pipeline and not necessarily a command that's on one physical line. Not all commands that are on one physical line are one-liners.

Even though the following command is on more than one physical line, it's a PowerShell one-liner because it's one continuous pipeline. It could be written on one physical line, but I've chosen to line break at the pipe symbol which is one of the characters in PowerShell where a natural line break can occur.

```
PS C:\> Get-Service |
>> Where-Object CanPauseAndContinue -eq $true |
>> Select-Object -Property *
```

```
Name                : LanmanWorkstation
RequiredServices    : {NSI, MRxSmb20, Bowser}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Workstation
DependentServices   : {SessionEnv, Netlogon, Browser}
MachineName         : .
ServiceName         : LanmanWorkstation
ServicesDependedOn  : {NSI, MRxSmb20, Bowser}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Automatic
Site                :
Container           :

Name                : Netlogon
RequiredServices    : {LanmanWorkstation}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Netlogon
DependentServices   : {}
MachineName         : .
ServiceName         : Netlogon
ServicesDependedOn  : {LanmanWorkstation}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Automatic
Site                :
Container           :

Name                : vmicheartbeat
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Heartbeat Service
DependentServices   : {}
MachineName         : .
ServiceName         : vmicheartbeat
ServicesDependedOn  : {}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
```

```
StartType           : Manual
Site                :
Container           :

Name                : vmickvpexchange
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Data Exchange Service
DependentServices   : {}
MachineName         : .
ServiceName         : vmickvpexchange
ServicesDependedOn  : {}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Manual
Site                :
Container           :

Name                : vmicrdv
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Remote Desktop Virtualization Service
DependentServices   : {}
MachineName         : .
ServiceName         : vmicrdv
ServicesDependedOn  : {}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Manual
Site                :
Container           :

Name                : vmicshutdown
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Guest Shutdown Service
DependentServices   : {}
MachineName         : .
ServiceName         : vmicshutdown
```

```
ServicesDependedOn  : {}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Manual
Site                :
Container           :

Name                : vmictimesync
RequiredServices    : {VmGid}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Time Synchronization Service
DependentServices   : {}
MachineName         : .
ServiceName         : vmictimesync
ServicesDependedOn  : {VmGid}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Manual
Site                :
Container           :

Name                : vmicvss
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Volume Shadow Copy Requestor
DependentServices   : {}
MachineName         : .
ServiceName         : vmicvss
ServicesDependedOn  : {}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Manual
Site                :
Container           :

Name                : Winmgmt
RequiredServices    : {RPCSS}
CanPauseAndContinue : True
CanShutdown         : True
CanStop             : True
```

```
DisplayName         : Windows Management Instrumentation
DependentServices   : {wscsvc, NcaSvc, iphlpsvc}
MachineName         : .
ServiceName         : Winmgmt
ServicesDependedOn  : {RPCSS}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Automatic
Site                :
Container           :

PS C:\>
```

Other characters where natural line breaks can occur that are commonly used include the comma, and opening brackets, braces, and parenthesis. Others that aren't so common include the semicolon, equals sign, and both opening single quotes and double quotes.

Using the backtick or grave accent character as a line continuation character is a controversial topic, but my recommendation is to try to avoid it if at all possible. I see PowerShell commands written all the time where a backtick is used immediately after a natural line break character and there's simply no reason for it to be there.

```
PS C:\> Get-Service -Name w32time |
>> Select-Object -Property *


Name                : w32time
RequiredServices    : {}
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
DisplayName         : Windows Time
DependentServices   : {}
MachineName         : .
ServiceName         : w32time
ServicesDependedOn  : {}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
StartType           : Manual
Site                :
Container           :

PS C:\>
```

The commands shown in the previous two examples work fine in the PowerShell console, but if you try to run them in the console pane of the PowerShell ISE, they'll generate an error. This is because the console pane of the PowerShell ISE doesn't wait for the remainder of the command to be entered on the next line like the PowerShell console does. To alleviate this problem, use shift + enter instead of just pressing enter when continuing a command on another line in the console pane of the PowerShell ISE.

Now for an example of a command that's on one physical line, but not a PowerShell one-liner because it's not one continuous pipeline. It's two separate commands placed on one line and separated by a semicolon.

```
PS C:\> $Service = 'w32time'; Get-Service -Name $Service


Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time

PS C:\>
```

Many programming and scripting languages require a semicolon at the end of each line and while they can be used that way in PowerShell, it's not recommended because they're simply not needed.

Filtering Left

The results of the commands shown in this chapter have been filtered down to a subset. For example, Get-Service was used with the Name parameter to filter the list of services that were returned to only the Windows Time service.

You always want to filter the results down to what you're looking for as early as possible in the pipeline. Best case scenario, this is accomplished by using parameters of the first command that's specified or the one to the far left which is why it's sometimes called filtering left.

The following example uses the Name parameter of Get-Service to immediately filter down the results to only the Windows Time service.

```
PS C:\> Get-Service -Name w32time
```

```
Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time

PS C:\>
```

It's not uncommon to see examples where the command is piped to Where-Object to perform the filtering.

```
PS C:\> Get-Service | Where-Object Name -eq w32time


Status   Name               DisplayName
------   ----               -----------
Running  W32Time            Windows Time

PS C:\>
```

The first option performs the filtering at the source and only returns the results for the Windows Time service. The second option returns all of the services only to pipe them to another command to perform the filtering. While this may not seem like a big deal in this example, imagine if you were querying a list of Active Directory users. Do you really want to pull the information for thousands, tens of thousand, or possibly hundreds of thousands of user accounts from Active Directory only to pipe them to another command which filters them down to a subset? My recommendation is to always filter left even when it doesn't seem to matter and you'll be so use to it that you'll automatically filter left when it really does matter.

I once had someone tell me that the order you specify the commands in doesn't matter. That couldn't be further from the truth. The order that the commands are specified in does indeed matter when performing filtering. For example, if you're using Select-Object to select only a few properties, but need to perform filtering with Where-Object on properties that won't be in the selection. In that scenario, the filtering must occur first, otherwise the property wouldn't exist in the pipeline when trying to perform the filtering.

```
PS C:\> Get-Service |
Select-Object -Property DisplayName, Running, Status |
Where-Object CanPauseAndContinue
```

The command in the previous example won't return any results because the CanStopAndContinue property doesn't exist when the results of Select-Object are piped to Where-Object. That particular property wasn't "selected" and in essence, it was filtered out. Reversing the order of Select-Object and Where-Object produces the desired results.

```
PS C:\> Get-Service |
Where-Object CanPauseAndContinue |
Select-Object -Property DisplayName, Running, Status
```

```
DisplayName                                   Running  Status
-----------                                   -------  ------
Workstation                                           Running
Netlogon                                              Running
Hyper-V Heartbeat Service                             Running
Hyper-V Data Exchange Service                         Running
Hyper-V Remote Desktop Virtualization Service         Running
Hyper-V Guest Shutdown Service                        Running
Hyper-V Time Synchronization Service                  Running
Hyper-V Volume Shadow Copy Requestor                  Running
Windows Management Instrumentation                    Running

PS C:\>
```

The Pipeline

As you've seen in many of the examples shown so far throughout this book, many times the output of one command can be used as input for another command. In chapter 3, Get-Member was used to determine what type of object a command produces. Chapter 3 also showed using the ParameterType parameter of Get-Command to determine what commands accepted that type of input, although not necessarily by pipeline input.

Depending on how thorough a commands help is, it may include an INPUTS and OUTPUTS section.

```
PS C:\> help Stop-Service -Full
```

```
INPUTS
    System.ServiceProcess.ServiceController, System.String
        You can pipe a service object or a string that contains the name of a service
        to this cmdlet.

OUTPUTS
    None, System.ServiceProcess.ServiceController
        This cmdlet generates a System.ServiceProcess.ServiceController object that
        represents the service, if you use the PassThru parameter. Otherwise, this
        cmdlet does not generate any output.
```

Only the relevant section of the help is shown in the previous results. As you can see, the INPUTS section states that a ServiceController or a String object can be piped to the Stop-Service cmdlet. It doesn't tell you which parameters accept that type of input. One of the easiest ways to determine that information is to look through the different parameters in the full version of the help for the Stop-Service cmdlet.

```
PS C:\> help Stop-Service -Full


-DisplayName <String[]>
    Specifies the display names of the services to stop. Wildcard characters are
    permitted.

    Required?                    true
    Position?                    named
    Default value                None
    Accept pipeline input?       False
    Accept wildcard characters?  false

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

    Required?                    true
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByValue)
    Accept wildcard characters?  false

-Name <String[]>
    Specifies the service names of the services to stop. Wildcard characters are
    permitted.

    Required?                    true
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByPropertyName, ByValue)
    Accept wildcard characters?  false

PS C:\>
```

Once again, I've only shown the relevant portion of the help in the previous set of results. Notice that the DisplayName parameter doesn't accept pipeline input, the InputObject parameter accepts pipeline input by value for ServiceController objects, and the Name parameter accepts pipeline input by value for string objects. It also accepts pipeline input by property name.

When a parameter accepts pipeline input by both property name and by value, it always tries by value first and if by value fails, then and only then does it try by property name. By value is a little misleading. I prefer to call by value "by type." This means if you pipe the results of a command that produces a ServiceController object type to Stop-Service, it will bind that input to the InputObject parameter, but if you pipe the results of a command that produces string output to Stop-Service, it will bind it to the Name parameter. If you pipe the results of a command that doesn't produce a ServiceController or String object to Stop-Service, but it does produce output that contains a property called "Name", then and only then will it bind the Name property from the output to the Name parameter as input for the Stop-Service command.

Determine what type of output the Get-Service command produces.

```
PS C:\> Get-Service -Name w32time | Get-Member


   TypeName: System.ServiceProcess.ServiceController
```

Get-Service produces a ServiceController object type.

As you previously saw in the help, the InputObject parameter of Stop-Service accepts ServiceController objects via the pipeline by value (by type). This means that when the results of the Get-Service cmdlet are piped to Stop-Service, they bind to the InputObject parameter of Stop-Service.

```
PS C:\> Get-Service -Name w32time | Stop-Service
```

Now to try string input. Pipe w32time to Get-Member just to confirm that it's a string.

```
PS C:\> 'w32time' | Get-Member


   TypeName: System.String
```

As previously shown in the help, piping a string to Stop-Service will bind it to the Name parameter of Stop-Service by value (by type). Test this by piping w32time to Stop-Service.

```
PS C:\> 'w32time' | Stop-Service
```

Notice that in the previous example, I used single quotes around the string "w32time." You should always use single quotes instead of double quotes unless the contents of the quoted items contain a variable which needs to be expanded to its actual value. By using single quotes, PowerShell doesn't have to parse the contents contained within the quotes so your code will run a little faster.

Create a custom object to test pipeline input by property name for the Name parameter of Stop-Service.

```
PS C:\> $CustomObject = [pscustomobject]@{
>> Name = 'w32time'
>> }
```

The contents of the CustomObject variable is a PSCustomObject object type and it contains a property named “Name”.

```
PS C:\> $CustomObject | Get-Member


   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
Name        NoteProperty string Name=w32time

PS C:\>
```

If you were to surround the CustomObject variable with quotes in the previous example, you would want to use double quotes otherwise if single quotes were used, the literal $CustomObject would be piped to Get-Member instead of the value it contains.

Although piping the contents of the CustomObject variable to Stop-Service cmdlet binds to the Name parameter, this time it binds by property name instead of by value because the contents of the CustomObject variable aren’t a string, but they do contain a property named “Name”.

Create another custom object except this time use another name such as “service”.

```
PS C:\> $CustomObject = [pscustomobject]@{
>> Service = 'w32time'
>> }
```

An error is generated when trying to pipe the contents of this updated CustomObject variable to Stop-Service because it doesn’t produce a ServiceController or String object and it doesn’t have a property named “Name”.

```
PS C:\> $CustomObject | Stop-Service
```

```
Stop-Service : Cannot find any service with service name '@{Service=w32time}'.
At line:1 char:17
+ $CustomObject | Stop-Service
+
    + CategoryInfo          : ObjectNotFound: (@{Service=w32time}:String) [Stop-Service]
   , ServiceCommandException
    + FullyQualifiedErrorId : NoServiceFoundForGivenName,Microsoft.PowerShell.Commands.S
   topServiceCommand

PS C:\>
```

If the output of one command doesn't line up with the pipeline input options for another command as shown in the previous example, Select-Object can be used to rename the property to make the properties lineup correctly.

```
PS C:\> $CustomObject |
>> Select-Object -Property @{label='Name';Expression={$_.Service}} |
>> Stop-Service
PS C:\>
```

In the previous example, Select-Object was used to rename the Service property to a property named "Name" so that it could be accepted via pipeline input by the Stop-Service cmdlet.

The syntax of the previous example may seem a little complicated at first. What I have learned is that you'll never learn the syntax by copy and pasting code. If you take the time to type the code in, after a few times it will become second nature. Having multiple monitors is a huge benefit because you can display the example code on one screen and type it in on another one.

If for some reason a command doesn't accept pipeline input on a parameter that you want to provide input on from the output of another command, you can always use parameter input.

To demonstrate using the output of one command as parameter input for another, first save the display name for a couple of Windows services into a text file.

```
PS C:\> 'Background Intelligent Transfer Service', 'Windows Time' |
Out-File -FilePath $env:TEMP\services.txt
```

Simple run the command that you want to provide the output from within parenthesis as the value for the parameter of the command to provide the input for, or Stop-Service in this scenario as shown in the following example.

```
PS C:\> Stop-Service -DisplayName (Get-Content -Path $env:TEMP\services.txt)
```

This is just like order of operations in Algebra for those of you who remember how it works. The portion of the command within parenthesis will always run prior to the outer portion of the command.

PowerShellGet

PowerShellGet is a PowerShell module that contains commands for discovering, installing, publishing, and updating PowerShell modules and other artifacts to/from a NuGet repository. PowerShellGet ships with PowerShell version 5.0 and higher. It is available as a separate download for PowerShell version 3.0 and higher.

Microsoft hosts an online NuGet repository called the PowerShell Gallery. Although this repository is hosted by Microsoft, the majority of the PowerShell modules and code contained within the repository isn't written by Microsoft and should be thoroughly reviewed in an isolated test environment before being considered suitable for use in a production environment.

Most companies will want to host their own internal private NuGet repository where they can post their internal use only modules as well as modules that they've downloaded from other sources once they've validated them as being non-malicious.

Use the Find-Module cmdlet that's part of the PowerShellGet module to find a module in the PowerShell Gallery that I wrote named MrToolkit.

```
PS C:\> Find-Module -Name MrToolkit
```

```
NuGet provider is required to continue
PowerShellGet requires NuGet provider version '2.8.5.201' or newer to interact with
NuGet-based repositories. The NuGet provider must be available in 'C:\Program
Files\PackageManagement\ProviderAssemblies' or
'C:\Users\MrAdmin\AppData\Local\PackageManagement\ProviderAssemblies'. You can also
install the NuGet provider by running 'Install-PackageProvider -Name NuGet
-MinimumVersion 2.8.5.201 -Force'. Do you want PowerShellGet to install and import the
NuGet provider now?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):

Version    Name                                Repository           Description
-------    ----                                ----------           -----------
1.1        MrToolkit                           PSGallery            Misc PowerShell Tools

PS C:\>
```

As shown in the previous example, the first time you use one of the commands from the PowerShellGet module, you'll be prompted to install the NuGet provider.

To install the MrToolkit module, simply pipe the previous command to Install-Module.

```
PS C:\> Find-Module -Name MrToolkit | Install-Module
```

```
Untrusted repository
You are installing the modules from an untrusted repository. If you trust this
repository, change its InstallationPolicy value by running the Set-PSRepository cmdlet.
Are you sure you want to install the modules from
'https://www.powershellgallery.com/api/v2/'?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"): y
PS C:\>
```

Since the PowerShell Gallery is an untrusted repository, it prompts you to approve the installation of the module.

Finding Pipeline Input - The Easy Way

The MrToolkit module contains a function named Get-MrPipelineInput that can be used to easily determine what parameters of a command accept pipeline input and what type of object they accept as well as if they accept pipeline input by value or by property name.

```
PS C:\> Get-MrPipelineInput -Name Stop-Service

ParameterName ParameterType                              ValueFromPipeline ValueFromPipelineByP\
ropertyName
------------- -------------                              ----------------- ---------------
InputObject   System.ServiceProcess.ServiceController[]              True           False
Name          System.String[]                                        True            True

PS C:\>
```

As you can see in the previous set of results, the same information we previously determined by sifting through the help can easily be determined with this function.

Summary

In this chapter, you've learned about PowerShell one-liners. You've learned that the number of physical lines that a command is on has nothing to do with whether or not it's a PowerShell one-liner. You've also learned about filtering left, the pipeline, and PowerShellGet.

Review

1. What is a PowerShell one-liner?
2. What are some of the characters where natural line breaks can occur in PowerShell?
3. Why should you filter left?
4. What are the two ways that a PowerShell command can accept pipeline input?
5. Why shouldn't you trust commands found in the PowerShell Gallery?

Recommended Reading

- about_Pipelines
- about_Command_Syntax
- about_Parameters
- PowerShellGet: The BIG EASY way to discover, install, and update PowerShell modules

Chapter 5

Requirements

The SQL Server PowerShell module which installs as part of SSMS (SQL Server Management Studio) 2016 is required by some of the examples shown in this chapter. It will also be used in subsequent chapters. Download and install it on your Windows 10 lab environment computer.

Format Right

In chapter 4, you learned to filter as far to the left as possible. The rule for manually formatting a command's output is similar to that rule except it needs to occur as far to the right as possible.

The most common format commands are Format-Table and Format-List. Format-Wide and Format-Custom can also be used, but are less common.

As mentioned in Chapter 3, a command that returns five or more properties defaults to a list unless custom formatting is used.

```
PS C:\> Get-Service -Name w32time | Select-Object -Property Status, DisplayName, Can*


Status              : Running
DisplayName         : Windows Time
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True

PS C:\>
```

Use the Format-Table cmdlet to manually override the formatting and show the output in a table instead of a list.

```
PS C:\> Get-Service -Name w32time | Select-Object -Property Status, DisplayName, Can* |
Format-Table
```

```
 Status DisplayName  CanPauseAndContinue CanShutdown CanStop
 ------ -----------  ------------------- ----------- -------
Running Windows Time               False        True    True

PS C:\>
```

The default output for Get-Service is three properties in a table.

```
PS C:\> Get-Service -Name w32time


Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time

PS C:\>
```

Use the Format-List cmdlet to override the default formatting and return the results in a list.

```
PS C:\> Get-Service -Name w32time | Format-List


Name                : w32time
DisplayName         : Windows Time
Status              : Running
DependentServices   : {}
ServicesDependedOn  : {}
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
ServiceType         : Win32ShareProcess

PS C:\>
```

Notice that simply piping Get-Service to Format-List made it return additional properties. This doesn't occur with every command and is actually due to the way the formatting for that particular command is setup behind the scenes.

The number one thing to be aware of with the format cmdlets is they produce format objects which are different than normal objects in PowerShell.

```
PS C:\> Get-Service -Name w32time | Format-List | Get-Member
```

```
   TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatStartData

Name                                    MemberType Definition
----                                    ---------- ----------
Equals                                  Method     bool Equals(System.Object obj)
GetHashCode                             Method     int GetHashCode()
GetType                                 Method     type GetType()
ToString                                Method     string ToString()
autosizeInfo                            Property   Microsoft.PowerShell.Commands.Inter...
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef21dd47e99d3c9...
groupingEntry                           Property   Microsoft.PowerShell.Commands.Inter...
pageFooterEntry                         Property   Microsoft.PowerShell.Commands.Inter...
pageHeaderEntry                         Property   Microsoft.PowerShell.Commands.Inter...
shapeInfo                               Property   Microsoft.PowerShell.Commands.Inter...

   TypeName: Microsoft.PowerShell.Commands.Internal.Format.GroupStartData

Name                                    MemberType Definition
----                                    ---------- ----------
Equals                                  Method     bool Equals(System.Object obj)
GetHashCode                             Method     int GetHashCode()
GetType                                 Method     type GetType()
ToString                                Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef21dd47e99d3c9...
groupingEntry                           Property   Microsoft.PowerShell.Commands.Inter...
shapeInfo                               Property   Microsoft.PowerShell.Commands.Inter...

   TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData

Name                                    MemberType Definition
----                                    ---------- ----------
Equals                                  Method     bool Equals(System.Object obj)
GetHashCode                             Method     int GetHashCode()
GetType                                 Method     type GetType()
ToString                                Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef21dd47e99d3c9...
formatEntryInfo                         Property   Microsoft.PowerShell.Commands.Inter...
outOfBand                               Property   bool outOfBand {get;set;}
writeStream                             Property   Microsoft.PowerShell.Commands.Inter...

   TypeName: Microsoft.PowerShell.Commands.Internal.Format.GroupEndData

Name                                    MemberType Definition
----                                    ---------- ----------
Equals                                  Method     bool Equals(System.Object obj)
GetHashCode                             Method     int GetHashCode()
GetType                                 Method     type GetType()
```

```
ToString                                Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef21dd47e99d3c9...
groupingEntry                           Property   Microsoft.PowerShell.Commands.Inter...

   TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatEndData

Name                                    MemberType Definition
----                                    ---------- ----------
Equals                                  Method     bool Equals(System.Object obj)
GetHashCode                             Method     int GetHashCode()
GetType                                 Method     type GetType()
ToString                                Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef21dd47e99d3c9...
groupingEntry                           Property   Microsoft.PowerShell.Commands.Inter...

PS C:\>
```

What this means is format commands can't be piped to most other commands. They can be piped to some of the Out-* commands, but that's about it. This is why you want to perform any formatting at the very end of the line (format right).

Aliases

An alias in PowerShell is a another shorter name for a command. PowerShell includes a set of built-in aliases and you can also define your own aliases.

The Get-Alias cmdlet is used to find aliases. If you already know the alias for a command, the Name parameter is used to determine what command the alias is associated with.

```
PS C:\> Get-Alias -Name gcm


CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           gcm -> Get-Command

PS C:\>
```

Multiple aliases can be specified for the value of the Name parameter.

```
PS C:\> Get-Alias -Name gcm, gm
```

```
CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           gcm -> Get-Command
Alias           gm -> Get-Member

PS C:\>
```

You'll often see the Name parameter omitted since it's a positional parameter.

```
PS C:\> Get-Alias gm

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           gm -> Get-Member

PS C:\>
```

The previous examples are most commonly used when you see someone using an alias that you're not familiar with and you're trying to figure out what the actual command is that they're using.

If you want to find aliases for a command, you'll need to use the Definition parameter.

```
PS C:\> Get-Alias -Definition Get-Command, Get-Member

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           gcm -> Get-Command
Alias           gm -> Get-Member

PS C:\>
```

The Definition parameter can't be used positionally so it must be specified.

Aliases can save you a few keystrokes and while they're fine when you're typing commands into the console, they shouldn't be used in scripts or any code that you're saving or sharing with others. As mentioned earlier in this book, using full cmdlet and parameter names are more self-documenting and they're easier to understand.

Use caution when creating your own aliases because they'll only exist in your current PowerShell session on your computer.

Providers

A provider in PowerShell is an interface that allows file system like access to a datastore. There are a number of built-in providers in PowerShell.

```
 PS C:\> Get-PSProvider


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

PS C:\>
```

As you can see in the previous results, there are built-in providers for the registry, aliases, environment variables, the file system, functions, variables, certificates, and wsman.

The actual drives that these providers use to expose their datastore can be determined with the Get-PSDrive cmdlet. The Get-PSDrive cmdlet not only displays drives exposed by providers, but it also displays Windows logical drives including drives mapped to network shares.

```
PS C:\> Get-PSDrive


Name           Used (GB)     Free (GB) Provider      Root
----           ---------     --------- --------      ----
A                                      FileSystem    A:\
Alias                                  Alias
C                  14.41        112.10 FileSystem    C:\
Cert                                   Certificate   \
D                                      FileSystem    D:\
Env                                    Environment
Function                               Function
HKCU                                   Registry      HKEY_CURRENT_USER
HKLM                                   Registry      HKEY_LOCAL_MACHINE
Variable                               Variable
WSMan                                  WSMan

PS C:\>
```

Third party modules, such as the Active Directory PowerShell module which installs as part of the RSAT and the SQLServer PowerShell module which installs as part of SSMS 2016, each add their own PowerShell provider and PSDrive.

Import the Active Directory and SQL Server PowerShell modules.

```
PS C:\> Import-Module -Name ActiveDirectory, SQLServer
```

Check to see if any additional PowerShell providers were added.

```
PS C:\> Get-PSProvider


Name                 Capabilities                       Drives
----                 ------------                       ------
Registry             ShouldProcess, Transactions        {HKLM, HKCU}
Alias                ShouldProcess                      {Alias}
Environment          ShouldProcess                      {Env}
FileSystem           Filter, ShouldProcess, Credentials {C, A, D}
Function             ShouldProcess                      {Function}
Variable             ShouldProcess                      {Variable}
ActiveDirectory      Include, Exclude, Filter, Shoul... {AD}
SqlServer            Credentials                        {SQLSERVER}

PS C:\>
```

Notice that in the previous set of results, two new PowerShell providers now exist, one for Active Directory and another one for SQL Server.

A PSDrive for each of those modules was also added.

```
PS C:\> Get-PSDrive


Name           Used (GB)     Free (GB) Provider      Root
----           ---------     --------- --------      ----
A                                      FileSystem    A:\
AD                                     ActiveDire... //RootDSE/
Alias                                  Alias
C                  19.38        107.13 FileSystem    C:\
Cert                                   Certificate   \
D                                      FileSystem    D:\
Env                                    Environment
Function                               Function
HKCU                                   Registry      HKEY_CURRENT_USER
HKLM                                   Registry      HKEY_LOCAL_MACHINE
SQLSERVER                              SqlServer     SQLSERVER:\
Variable                               Variable
WSMan                                  WSMan

PS C:\>
```

PSDrives can be accessed just like a traditional file system.

```
PS C:\> Get-ChildItem -Path Cert:\LocalMachine\CA


   PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\CA

Thumbprint                                Subject
----------                                -------
FEE449EE0E3965A5246F000E87FDE2A065FD89D4  CN=Root Agency
D559A586669B08F46A30A133F8A9ED3D038E2EA8  OU=www.verisign.com/CPS Incorp.by Ref. LIABI...
109F1CAED645BB78B3EA2B94C0697C740733031C  CN=Microsoft Windows Hardware Compatibility,...

PS C:\>
```

Comparison Operators

PowerShell contains a number of comparison operators which are used to compare values and to find values that match certain patterns. Table 5-1 contains a list of comparison operators in PowerShell.

-eq	Equal to.
-ne	Not equal to.
-gt	Greater than.
-ge	Greater than or equal to.
-lt	Less than.
-le	Less than or equal to.
-Like	Match using the * wildcard character.
-NotLike	Does not match using the * wildcard character.
-Match	Matches the specified regular expression.
-NotMatch	Does not match the specified regular expression.
-Contains	Determines if a collection contains a specified value.
-NotContains	Determines if a collection does not contain a specific value.
-In	Determines if a specified value is in a collection.
-NotIn	Determines if a specified value is not in a collection.
-Replace	Replaces the specified value.

Table 5-1

All of the operators listed in Table 5-1 are case-insensitive. Simply place a “c” in front of the operator listed in Table 5-1 to make it case-sensitive. For example, “-ceq” is the case-sensitive version of the “-eq” comparison operator.

Proper case PowerShell is equal to lower case powershell using the equals comparison operator.

```
PS C:\> 'PowerShell' -eq 'powershell'
```

```
True
PS C:\>
```

It's not equal using the case-sensitive version of the equals comparison operator.

```
PS C:\> 'PowerShell' -ceq 'powershell'
```

```
False
PS C:\>
```

The not equal comparison operator reverses the condition.

```
PS C:\> 'PowerShell' -ne 'powershell'
```

```
False
PS C:\>
```

Greater than, greater than or equal to, less than, and less than or equal to are all designed for working with numeric values.

```
PS C:\> 5 -gt 5
```

```
False
PS C:\>
```

Using greater than or equal to instead of greater than with the previous example returns the Boolean true since five is equal to five.

```
PS C:\> 5 -ge 5
```

```
True
PS C:\>
```

Based on the results from the previous two examples, you can probably guess how both less than and less than or equal to work.

```
PS C:\> 5 -lt 10


True
PS C:\>
```

The Like and Match operators can be confusing, even for experienced PowerShell users. Like is used with wildcard characters such as * and ? to perform "like" matches.

```
PS C:\> 'PowerShell' -like '*shell'


True
PS C:\>
```

Match uses a regular expression to perform the matching.

```
PS C:\> 'PowerShell' -match '^*.shell$'


True
PS C:\>
```

Use the range operator to store the numbers one through ten in a variable.

```
PS C:\> $Numbers = 1..10


PS C:\>
```

Determine if the Numbers variable includes fifteen.

```
PS C:\> $Numbers -contains 15


False
PS C:\>
```

Determine if it includes the number ten.

```
PS C:\> $Numbers -contains 10
```

```
True
PS C:\>
```

NotContains reverses the logic to see if the Numbers variable doesn't contain a value.

```
PS C:\> $Numbers -notcontains 15


True
PS C:\>
```

The previous example returns the Boolean true because it's true that the Numbers variable doesn't contain fifteen. It does however contain the number ten so it's false when it's tested to see if it doesn't contain ten.

```
PS C:\> $Numbers -notcontains 10


False
PS C:\>
```

The "in" comparison operator was first introduced in PowerShell version 3.0. It's used to determine if a value is "in" an array. The Numbers variable is an array since it contains multiple values.

```
PS C:\> 15 -in $Numbers


False
PS C:\>
```

In other words, "in" performs the same test as the contains comparison operator except from the opposite direction.

```
PS C:\> 10 -in $Numbers


True
PS C:\>
```

Fifteen is not in the Numbers array so false is returned in the following example.

```
PS C:\> 15 -in $Numbers


False
PS C:\>
```

Just like the contains operator, not reverses the logic for the "in" operator.

```
PS C:\> 10 -notin $Numbers


False
PS C:\>
```

The previous example returns false because the Numbers array does include ten and the condition was testing to determine if it didn't contain ten.

Fifteen is "not in" the Numbers array so it returns the Boolean true.

```
PS C:\> 15 -notin $Numbers


True
PS C:\>
```

The replace operator does just want you would think. It's used to replace something. Specifying one value replaces that particular value with nothing as shown in the following example where I'll replace Shell with nothing.

```
PS C:\> 'PowerShell' -replace 'Shell'


Power
PS C:\>
```

If you want to replace a value with a different value, specify the new value in the second position after what you want to replace. SQL Saturday in Baton Rouge is an event that I try to speak at every year. In the following example, I'll replace the word "Saturday" with the abbreviation "Sat".

```
PS C:\> 'SQL Saturday - Baton Rouge' -Replace 'saturday','Sat'
```

```
SQL Sat - Baton Rouge
PS C:\>
```

There are also methods such as the Replace method which can be used to replace things very similar to the way the replace operator works. However, while the Replace operator is case-insensitive by default, the Replace method is case-sensitive.

```
PS C:\> 'SQL Saturday - Baton Rouge'.Replace('saturday','Sat')
```

```
SQL Saturday - Baton Rouge
PS C:\>
```

Notice that the word Saturday was not replaced in the previous example. This is because it was specified in a different case than the original. When the word Saturday is specified in the same case as the original, the Replace method does indeed perform the replace as expected.

```
PS C:\> 'SQL Saturday - Baton Rouge'.Replace('Saturday','Sat')
```

```
SQL Sat - Baton Rouge
PS C:\>
```

Be very careful when using methods to transform data because you can run into unforeseen problems such as failing what's called the Turkey Test when simply trying to convert to upper case as discussed in my blog article titled "Using Pester to Test PowerShell Code with Other Cultures". My recommendation is to use an operator instead of a method whenever possible to avoid these types of problems altogether.

While the comparison operators can be used as shown in the previous examples, I normally find myself using them with the Where-Object cmdlet to perform some type of filtering.

Summary

In this chapter, you've learned a number of different topics to include Formatting Right, Aliases, Providers, and Comparison Operators.

Review

1. Why is it necessary to perform Formatting as far to the right as possible?
2. How do you determine what the actual cmdlet is for the "%" alias?
3. Why shouldn't you use aliases in scripts you save or code you share with others?
4. Perform a directory listing on the drives that are associated with one of the registry providers.
5. What's one of the main benefits of using the replace operator instead of the replace method?

Recommended Reading

- Format-Table
- Format-List
- Format-Wide
- about_Aliases
- about_Providers
- about_Comparison_Operators

about_Arrays

Chapter 6

Scripting

When you move from writing PowerShell one-liners to writing scripts, it sounds a lot more complicated than it really is. A script is nothing more than the same or similar commands that you would run interactively in the PowerShell console, except they're saved as a PS1 file. There are some scripting constructs that you may use such as a foreach loop instead of the ForEach-Object cmdlet. To beginners the differences can be confusing especially when you consider that foreach is both a scripting construct and an alias for the ForEach-Object cmdlet.

Looping

One of the great things about PowerShell is once you figure out how to do something for one item such as one Active Directory user account, it's almost as easy to perform the same task for hundreds of items. Simply loop through the items with one of the many different types of loops in PowerShell.

ForEach-Object

ForEach-Object is a cmdlet for iterating through items inline such as with PowerShell one-liners. ForEach-Object streams the objects through the pipeline.

Although the Module parameter of Get-Command accepts multiple values that are strings, it only accepts them via pipeline input by property name or via parameter input. In the following scenario, if I want to pipe two strings by value to Get-Command for use with the Module parameter, I would need to use the ForEach-Object cmdlet.

```
PS C:\> 'ActiveDirectory', 'SQLServer' |
>> ForEach-Object {Get-Command -Module $_} |
>> Group-Object -Property ModuleName -NoElement |
>> Sort-Object -Property Count -Descending
```

```
Count Name
----- ----
  147 ActiveDirectory
   82 SqlServer

PS C:\>
```

In the previous example, $_ is the current object. Beginning with PowerShell version 3.0, $PSItem can be used instead of $_ but I find that most experienced PowerShell users still prefer using $_ since it's backwards compatible and less to type.

The foreach scripting construct stores all of the items in memory before it starts iterating through them which could cause some problems if you don't know how many items you're working with.

```
PS C:\> $ComputerName = 'DC01', 'WEB01'
PS C:\> foreach ($Computer in $ComputerName) {
>>     Get-ADComputer -Identity $Computer
>> }


DistinguishedName : CN=DC01,OU=Domain Controllers,DC=mikefrobbins,DC=com
DNSHostName       : dc01.mikefrobbins.com
Enabled           : True
Name              : DC01
ObjectClass       : computer
ObjectGUID        : c38da20c-a484-469d-ba4c-bab3fb71ae8e
SamAccountName    : DC01$
SID               : S-1-5-21-2989741381-570885089-3319121794-1001
UserPrincipalName :

DistinguishedName : CN=WEB01,CN=Computers,DC=mikefrobbins,DC=com
DNSHostName       : web01.mikefrobbins.com
Enabled           : True
Name              : WEB01
ObjectClass       : computer
ObjectGUID        : 33aa530e-1e31-40d8-8c78-76a18b673c33
SamAccountName    : WEB01$
SID               : S-1-5-21-2989741381-570885089-3319121794-1107
UserPrincipalName :

PS C:\>
```

Many times a loop such as foreach or ForEach-Object is necessary otherwise you'll receive an error message.

```
PS C:\> Get-ADComputer -Identity 'DC01', 'WEB01'


Get-ADComputer : Cannot convert 'System.Object[]' to the type
'Microsoft.ActiveDirectory.Management.ADComputer' required by parameter 'Identity'.
Specified method is not supported.
At line:1 char:26
+ Get-ADComputer -Identity 'DC01', 'WEB01'
+                          ~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-ADComputer], ParameterBindingExc
   eption
    + FullyQualifiedErrorId : CannotConvertArgument,Microsoft.ActiveDirectory.Management
   .Commands.GetADComputer

PS C:\>
```

Other times you can simply learn a little more about how the command works by viewing the help for it to retrieve the same results while eliminating the loop altogether.

```
PS C:\> 'DC01', 'WEB01' | Get-ADComputer


DistinguishedName : CN=DC01,OU=Domain Controllers,DC=mikefrobbins,DC=com
DNSHostName       : dc01.mikefrobbins.com
Enabled           : True
Name              : DC01
ObjectClass       : computer
ObjectGUID        : c38da20c-a484-469d-ba4c-bab3fb71ae8e
SamAccountName    : DC01$
SID               : S-1-5-21-2989741381-570885089-3319121794-1001
UserPrincipalName :

DistinguishedName : CN=WEB01,CN=Computers,DC=mikefrobbins,DC=com
DNSHostName       : web01.mikefrobbins.com
Enabled           : True
Name              : WEB01
ObjectClass       : computer
ObjectGUID        : 33aa530e-1e31-40d8-8c78-76a18b673c33
SamAccountName    : WEB01$
SID               : S-1-5-21-2989741381-570885089-3319121794-1107
UserPrincipalName :

PS C:\>
```

As you can see in the previous examples, the Identity parameter for Get-ADComputer only accepts a single value when provided via parameter input, but it allows for multiple items when the input is provided via pipeline input.

For

A For loop iterates through an array while a specified condition is true. The For loop is not something that I use often, but it does have its uses.

```
PS C:\> for ($i = 1; $i -lt 5; $i++) {
>>     Write-Output "Sleeping for $i seconds"
>>     Start-Sleep -Seconds $i
>> }
```

```
Sleeping for 1 seconds
Sleeping for 2 seconds
Sleeping for 3 seconds
Sleeping for 4 seconds
PS C:\>
```

In the previous example, the loop will iterate four times by starting off with the number one and continue as long as the counter variable "i" is less than five. It will sleep for a total of ten seconds.

Do

There are two different Do loops in PowerShell. Do Until runs while the specified condition is false.

```
PS C:\> $number = Get-Random -Minimum 1 -Maximum 10
PS C:\> do {
>>     $guess = Read-Host -Prompt "What's your guess?"
>>     if ($guess -lt $number) {
>>         Write-Output 'Too low!'
>>     }
>>     elseif ($guess -gt $number) {
>>         Write-Output 'Too high!'
>>     }
>> }
>> until ($guess -eq $number)
```

```
What's your guess?: 1
Too low!
What's your guess?: 2
Too low!
What's your guess?: 3
PS C:\>
```

The previous example is a numbers game which will continue until the numeric value you guess equals the same number that the Get-Random cmdlet generated.

Do While is just the opposite, it runs as long as the specified condition is evaluated to true.

```
PS C:\> $number = Get-Random -Minimum 1 -Maximum 10
PS C:\> do {
>>     $guess = Read-Host -Prompt "What's your guess?"
>>     if ($guess -lt $number) {
>>         Write-Output 'Too low!'
>>     }
>>     elseif ($guess -gt $number) {
>>         Write-Output 'Too high!'
>>     }
>> }
>> while ($guess -ne $number)
```

```
What's your guess?: 1
Too low!
What's your guess?: 2
Too low!
What's your guess?: 3
Too low!
What's your guess?: 4
PS C:\>
```

The same results are achieved with a Do While loop by simply reversing the test condition to not equals.

Do loops always run at least once because the condition is evaluated at the end of the loop.

While

Similar to the Do While loop, a While loop runs as long as the specified condition is true. The difference however, is that a While loop evaluates the condition at the top of the loop before any code is run so it doesn't run at all if the condition evaluates to false.

```
PS C:\> $date = Get-Date -Date 'November 22'
PS C:\> while ($date.DayOfWeek -ne 'Thursday') {
>>     $date = $date.AddDays(1)
>> }
PS C:\> Write-Output $date
```

```
Thursday, November 23, 2017 12:00:00 AM

PS C:\>
```

The previous example calculates what day Thanksgiving Day is on in the United States. It's always on the fourth Thursday of November, so the loop starts with the twenty-second day of November and adds a day while the day of the week is not equal to Thursday. If the twenty-second is a Thursday, the loop doesn't run at all.

Break, Continue, and Return

Break is designed to break out of a loop. It's also commonly used with the switch statement.

```
PS C:\> for ($i = 1; $i -lt 5; $i++) {
>>     Write-Output "Sleeping for $i seconds"
>>     Start-Sleep -Seconds $i
>>     break
>> }
```

```
Sleeping for 1 seconds
PS C:\>
```

The break statement shown in the previous example causes the loop to exit on the first iteration.

Continue is designed to skip to the next iteration of a loop.

```
PS C:\> while ($i -lt 5) {
>>     $i += 1
>>     if ($i -eq 3) {
>>         continue
>>     }
>>     Write-Output $i
>> }
```

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

The previous example will output the numbers one, two, four, and five. It will skip number three and continue with the next iteration of the loop. Similar to break, continue breaks out of the loop except only for the current iteration and continues with the next iteration instead of breaking out of the loop and stopping.

Return is designed to exit out of the existing scope.

```
PS C:\> $number = 1..10
PS C:\> foreach ($n in $number) {
>>     if ($n -ge 4) {
>>         Return $n
>>     }
>> }


4
PS C:\>
```

Notice that in the previous example, return outputs the first result and then exists out of the loop. A more thorough explanation of the result statement can be found in one of my blog articles: "The PowerShell return keyword".

Summary

In this chapter, you've learned about the different types of loops that exist in PowerShell.

Review

1. What is the difference in the ForEach-Object cmdlet and the foreach scripting construct?
2. What is the primary advantage of using a While loop instead of a Do While or Do Until loop.
3. How do the break and continue statements differ?

Recommended Reading

- ForEach-Object
- about_ForEach
- about_For
- about_Do
- about_While
- about_Break
- about_Continue
- about_Return

Chapter 7 - Working with WMI

WMI and CIM

PowerShell ships by default with cmdlets for working with other technologies such as WMI (Windows Management Instrumentation). There are a number of native WMI cmdlets that exist in PowerShell without having to install any additional software or modules.

PowerShell has had cmdlets for working with WMI since its inception. Get-Command can be used to determine what WMI cmdlets exist in PowerShell. The following results are from my Windows 10 lab environment computer which is running PowerShell version 5.1. Your results may differ depending on what PowerShell version you're running.

```
PS C:\> Get-Command -Noun WMI*


CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Get-WmiObject                                      3.1.0.0    Microsof...
Cmdlet          Invoke-WmiMethod                                   3.1.0.0    Microsof...
Cmdlet          Register-WmiEvent                                  3.1.0.0    Microsof...
Cmdlet          Remove-WmiObject                                   3.1.0.0    Microsof...
Cmdlet          Set-WmiInstance                                    3.1.0.0    Microsof...

PS C:\>
```

CIM (Common Information Model) cmdlets were introduced in PowerShell version 3.0. The CIM cmdlets are designed so they can be used on both Windows and non-Windows machines. Moving forward, the WMI cmdlets will be deprecated so my recommendation is to use the CIM cmdlets instead of the older WMI ones.

The CIM cmdlets are all contained within a module. To obtain a list of the CIM cmdlets, simply use Get-Command with the Module parameter as shown in the following example.

```
PS C:\> Get-Command -Module CimCmdlets
```

```
CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Export-BinaryMiLog                                 1.0.0.0    CimCmdlets
Cmdlet          Get-CimAssociatedInstance                          1.0.0.0    CimCmdlets
Cmdlet          Get-CimClass                                       1.0.0.0    CimCmdlets
Cmdlet          Get-CimInstance                                    1.0.0.0    CimCmdlets
Cmdlet          Get-CimSession                                     1.0.0.0    CimCmdlets
Cmdlet          Import-BinaryMiLog                                 1.0.0.0    CimCmdlets
Cmdlet          Invoke-CimMethod                                   1.0.0.0    CimCmdlets
Cmdlet          New-CimInstance                                    1.0.0.0    CimCmdlets
Cmdlet          New-CimSession                                     1.0.0.0    CimCmdlets
Cmdlet          New-CimSessionOption                               1.0.0.0    CimCmdlets
Cmdlet          Register-CimIndicationEvent                        1.0.0.0    CimCmdlets
Cmdlet          Remove-CimInstance                                 1.0.0.0    CimCmdlets
Cmdlet          Remove-CimSession                                  1.0.0.0    CimCmdlets
Cmdlet          Set-CimInstance                                    1.0.0.0    CimCmdlets

PS C:\>
```

The CIM cmdlets still allow you to work with WMI so don't be confused when someone makes the statement "When I query WMI with the PowerShell CIM cmdlets."

As I previously mentioned, WMI is a separate technology from PowerShell and you're simply using the CIM cmdlets for accessing WMI. You may find an old VBScript that uses WQL (Windows Management Instrumentation Query Language) to query WMI such as in the following example.

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set colBIOS = objWMIService.ExecQuery _
    ("Select * from Win32_BIOS")

For each objBIOS in colBIOS
    Wscript.Echo "Manufacturer: " & objBIOS.Manufacturer
    Wscript.Echo "Name: " & objBIOS.Name
    Wscript.Echo "Serial Number: " & objBIOS.SerialNumber
    Wscript.Echo "SMBIOS Version: " & objBIOS.SMBIOSBIOSVersion
    Wscript.Echo "Version: " & objBIOS.Version
Next
```

You can take the WQL query from that VBScript and use it with the Get-CimInstance cmdlet in PowerShell without any modifications.

```
PS C:\> Get-CimInstance -Query 'Select * from Win32_BIOS'
```

```
SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 3810-1995-1654-4615-2295-2755-89
Version           : VRTUAL - 4001628

PS C:\>
```

That's not how I typically query WMI with PowerShell, but it does work and allows you to easily migrate existing VBScripts to PowerShell. If I start out writing a one-liner in PowerShell to query WMI, I'll use the following syntax.

```
PS C:\> Get-CimInstance -ClassName Win32_BIOS


SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 3810-1995-1654-4615-2295-2755-89
Version           : VRTUAL - 4001628

PS C:\>
```

If I only want the serial number, I can pipe the output to Select-Object and specify only the SerialNumber property.

```
PS C:\> Get-CimInstance -ClassName Win32_BIOS | Select-Object -Property SerialNumber


SerialNumber
------------
3810-1995-1654-4615-2295-2755-89

PS C:\>
```

By default there are a number of properties that are retrieved behind the scenes that are simply thrown away. While it may not matter much when querying WMI on the local computer, once you start querying remote computers, it's not only additional processing time to return that information, but also additional unnecessary information to have to pull across the network. Get-CimInstance has a Property parameter which can limit the information that's retrieved in order to minimize the network traffic if querying a remote computer. This makes the query to WMI more efficient.

```
PS C:\> Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber |
Select-Object -Property SerialNumber


SerialNumber
------------
3810-1995-1654-4615-2295-2755-89

PS C:\>
```

The previous results returned an object. To return a simple string, use the ExpandProperty parameter.

```
PS C:\> Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber |
Select-Object -ExpandProperty SerialNumber


3810-1995-1654-4615-2295-2755-89
PS C:\>
```

You could also use the dotted notation style of syntax to return a simple string which eliminates the need to pipe to Select-Object altogether.

```
PS C:\> (Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber).SerialNumber


3810-1995-1654-4615-2295-2755-89
PS C:\>
```

Query Remote Computers with the CIM cmdlets

I'm still running PowerShell as a local admin who is a domain user. When I try to query information from a remote computer using the Get-CimInstance cmdlet, I receive an access denied error message.

```
PS C:\> Get-CimInstance -ComputerName dc01 -ClassName Win32_BIOS
```

```
Get-CimInstance : Access is denied.
At line:1 char:1
+ Get-CimInstance -ComputerName dc01 -ClassName Win32_BIOS
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : PermissionDenied: (root\cimv2:Win32_BIOS:String) [Get-CimI
   nstance], CimException
    + FullyQualifiedErrorId : HRESULT 0x80070005,Microsoft.Management.Infrastructure.Cim
   Cmdlets.GetCimInstanceCommand
    + PSComputerName        : dc01

PS C:\>
```

Many people have security concerns when it comes to PowerShell, but the truth is you have exactly the same permissions with PowerShell as you do in the GUI, no more and no less. The problem in the previous example is the user I'm running PowerShell as doesn't have access to query information from WMI on the DC01 server. I could relaunch PowerShell as a domain administrator since Get-CimInstance doesn't have a Credential parameter, but trust me that isn't a good idea because then anything that I run from PowerShell would be running as a domain admin. That could be bad, worse, or maybe even ugly from a security standpoint depending on the situation.

Using the principle of least privilege, I elevate to my domain admin account on a per command basis using the Credential parameter if a command has one. Get-CimInstance doesn't have a Credential parameter so the solution in this scenario is to create a CimSession first and then use it instead of a computer name to query WMI on the remote computer.

```
PS C:\> $CimSession = New-CimSession -ComputerName dc01 -Credential (Get-Credential)
```

```
cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
PS C:\>
```

The CIM session was stored in a variable named CimSession as shown in the previous example. Notice that I also specified the Get-Credential cmdlet inside of parentheses, so it would be executed first and prompt me for alternate credentials before running the other portion of the command. I'll show you another more efficient way to specify alternate credentials later in this chapter, but it's important to understand this basic concept before making it more complicated.

The CIM session created in the previous example can now be used with the Get-CimInstance cmdlet to query the BIOS information from WMI on the remote computer.

```
PS C:\> Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS
```

```
SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 0986-6980-3916-0512-6608-8243-13
Version           : VRTUAL - 4001628
PSComputerName    : dc01

PS C:\>
```

There are several additional benefits to using CIM sessions instead of just specifying a computer name. When performing multiple queries to the same computer, creating a CIM session is much more efficient than what's required to setup and teardown a connection for each query. In other words, a CIM session only setups up the connection once and then numerous queries use that same session to retrieve information and then it's torn down once instead of setting it up and tearing it down with each individual query.

The Get-CimInstance cmdlet uses the WSMan protocol by default which means the remote computer needs PowerShell version 3.0 or higher in order for it to be able to connect. It's actually not the PowerShell version that matters, it's the stack version. The stack version can be determined using the Test-WSMan cmdlet. It needs to be version 3.0 and that's the version you'll find with PowerShell version 3.0 and higher.

```
PS C:\> Test-WSMan -ComputerName dc01


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

PS C:\>
```

The older WMI cmdlets use the DCOM protocol which is compatible with older versions of Windows, but it's typically blocked by the firewall on newer versions of Windows. There's a New-CimSessionOption cmdlet that allows you to create a DCOM protocol option for use with the New-CimSession cmdlet. This allows the Get-CimInstance cmdlet to be used to communicate with versions of Windows as far back as Windows Server 2000. This also means that PowerShell is not required on the remote computer when using the Get-CimInstance cmdlet with a CimSession that's configured to use the DCOM protocol.

Create the DCOM protocol option using the New-CimSessionOption cmdlet and store it in a variable.

```
PS C:\> $DCOM = New-CimSessionOption -Protocol Dcom
PS C:\>
```

Now back to the topic of making the entry of alternate credentials more efficient when they're specified on a command by command basis. Store your domain administrator or elevated credentials in a variable so you don't have to constantly enter them manually for each command.

```
PS C:\> $Cred = Get-Credential


cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
PS C:\>
```

I have a server named SQL03 which runs Windows Server 2008 (non-R2). It's the newest Windows Server operating system that doesn't have PowerShell installed by default.

Create a CimSession to SQL03 using the DCOM protocol.

```
PS C:\> $CimSession = New-CimSession -ComputerName sql03 -SessionOption $DCOM -Credential
$Cred


PS C:\>
```

Notice in the previous command, this time I specified the variable named Cred which contains my elevated credentials as the value for the Credential parameter instead of having to enter then manually again.

The output of the command to query WMI for BIOS information is the same regardless of the underlying protocol that's being used.

```
PS C:\> Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS


SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 7237-7483-8873-8926-7271-5004-86
Version           : VRTUAL - 4001628
PSComputerName    : sql03

PS C:\>
```

The Get-CimSession cmdlet is used to see what CimSessions are currently connected and what protocols they're using.

```
PS C:\> Get-CimSession


Id           : 1
Name         : CimSession1
InstanceId   : 80742787-e38e-41b1-a7d7-fa1369cf1402
ComputerName : dc01
Protocol     : WSMAN

Id           : 2
Name         : CimSession2
InstanceId   : 8fcabd81-43cf-4682-bd53-ccce1e24aecb
ComputerName : sql03
Protocol     : DCOM

PS C:\>
```

Retrieve and store both of the previously created CimSessions in a variable named CimSession.

```
PS C:\> $CimSession = Get-CimSession


PS C:\>
```

Query both of the computers with one command, one using the WSMan protocol and the other one with DCOM.

```
PS C:\> Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS


SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 0986-6980-3916-0512-6608-8243-13
Version           : VRTUAL - 4001628
PSComputerName    : dc01

SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 7237-7483-8873-8926-7271-5004-86
Version           : VRTUAL - 4001628
PSComputerName    : sql03

PS C:\>
```

I've written numerous blog articles about the WMI and CIM cmdlets. One of the most useful ones is about a function that I created to automatically determine if WSMan or DCOM should be used and setup the CIM session automatically without having to figure out which one manually. That blog article is titled "PowerShell Function to Create CimSessions to Remote Computers with Fallback to Dcom".

When you're finished with the CIM sessions, you should remove them with the Remove-CimSession cmdlet. To remove all CIM sessions, simply pipe Get-CimSession to Remove-CimSession.

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

```
PS C:\>
```

Summary

In this chapter you've learned about using PowerShell to work with WMI on both local and remote computers. You've also learned how to use the CIM cmdlets to work with remote computers with both the WSMan or DCOM protocol.

Review

1. What is the difference in the WMI and CIM cmdlets?
2. By default, what protocol does the Get-CimInstance cmdlet use?
3. What are some of the benefits of using a CIM session instead of specifying a computer name with Get-CimInstance?
4. How do you specify an alternate protocol other than the default one for use with Get-CimInstance?
5. How do you close or remove CIM sessions?

Recommended Reading

- about_WMI
- about_WMI_Cmdlets
- about_WQL
- CimCmdlets Module
- Video: Using CIM Cmdlets and CIM Sessions

Chapter 8 - PowerShell Remoting

There are many different ways to run commands against remote computers and servers with PowerShell. In the last chapter you saw how to remotely query WMI using the CIM cmdlets. There are also a number of cmdlets that have a built-in ComputerName parameter.

As shown in the following example, Get-Command can be used with the ParameterName parameter to determine what commands have a ComputerName parameter.

```
PS C:\> Get-Command -ParameterName ComputerName
```

```
CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Add-Computer                                       3.1.0.0    Microsof...
Cmdlet          Clear-EventLog                                     3.1.0.0    Microsof...
Cmdlet          Connect-PSSession                                  3.0.0.0    Microsof...
Cmdlet          Enter-PSSession                                    3.0.0.0    Microsof...
Cmdlet          Get-EventLog                                       3.1.0.0    Microsof...
Cmdlet          Get-HotFix                                         3.1.0.0    Microsof...
Cmdlet          Get-Process                                        3.1.0.0    Microsof...
Cmdlet          Get-PSSession                                      3.0.0.0    Microsof...
Cmdlet          Get-Service                                        3.1.0.0    Microsof...
Cmdlet          Get-WmiObject                                      3.1.0.0    Microsof...
Cmdlet          Invoke-Command                                     3.0.0.0    Microsof...
Cmdlet          Invoke-WmiMethod                                   3.1.0.0    Microsof...
Cmdlet          Limit-EventLog                                     3.1.0.0    Microsof...
Cmdlet          New-EventLog                                       3.1.0.0    Microsof...
Cmdlet          New-PSSession                                      3.0.0.0    Microsof...
Cmdlet          Receive-Job                                        3.0.0.0    Microsof...
Cmdlet          Receive-PSSession                                  3.0.0.0    Microsof...
Cmdlet          Register-WmiEvent                                  3.1.0.0    Microsof...
Cmdlet          Remove-Computer                                    3.1.0.0    Microsof...
Cmdlet          Remove-EventLog                                    3.1.0.0    Microsof...
Cmdlet          Remove-PSSession                                   3.0.0.0    Microsof...
Cmdlet          Remove-WmiObject                                   3.1.0.0    Microsof...
Cmdlet          Rename-Computer                                    3.1.0.0    Microsof...
Cmdlet          Restart-Computer                                   3.1.0.0    Microsof...
Cmdlet          Send-MailMessage                                   3.1.0.0    Microsof...
Cmdlet          Set-Service                                        3.1.0.0    Microsof...
Cmdlet          Set-WmiInstance                                    3.1.0.0    Microsof...
Cmdlet          Show-EventLog                                      3.1.0.0    Microsof...
Cmdlet          Stop-Computer                                      3.1.0.0    Microsof...
Cmdlet          Test-Connection                                    3.1.0.0    Microsof...
Cmdlet          Write-EventLog                                     3.1.0.0    Microsof...

PS C:\>
```

Commands such as Get-Process and Get-Service have a ComputerName parameter. This isn't the long term direction that Microsoft is heading with being able to run commands against remote computers. Even if you do find that a command you need to run against a remote computer does have a ComputerName parameter, chances are that you'll need to specify alternate credentials and it won't have a Credential parameter. Even if you decided to run PowerShell as an elevated account to work around that problem, a firewall between your computer and the remote computer or the firewall on the server itself will probably block the request.

In order to use the PowerShell remoting commands that are demonstrated from this point forward in this chapter, PowerShell version 2.0 or higher must exist on the remote computer and PowerShell remoting must be enabled on the remote computer. Table 8-1 shows the default setting of whether or not PowerShell remoting is enabled or disabled for the currently supported Microsoft operating systems.

Windows Operating System Version	**PowerShell Remoting**
Server 2008	Disabled
Windows 7	Disabled
Server 2008 R2	Disabled
Windows 8	Disabled
Server 2012	Enabled
Windows 8.1	Disabled
Server 2012 R2	Enabled
Windows 10	Disabled
Server 2016	Enabled

Table 8-1

PowerShell remoting can be enabled or re-enabled using the Enable-PSRemoting cmdlet.

```
PS C:\> Enable-PSRemoting
```

```
WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.

WinRM has been updated for remote management.
WinRM firewall exception enabled.

PS C:\>
```

One-To-One Remoting

If you need to run a PowerShell command against one remote computer and need it to be an interactive session, then one-to-one remoting is what you're after. This type of remoting is provided via the Enter-PSSession cmdlet.

In the last chapter, I stored my domain admin credentials in a variable named Cred. If you haven't already done so, go ahead and store your domain admin credentials in the Cred variable.

```
$Cred = Get-Credential
```

This allows you to enter the credentials one time and use them to elevate on a per command basis without having to enter them more than once, as long as your current PowerShell console or ISE session is active.

Create a one-to-one PowerShell remoting session to the domain controller named dc01.

```
PS C:\> Enter-PSSession -ComputerName dc01 -Credential $Cred
```

```
[dc01]: PS C:\Users\Administrator\Documents>
```

Notice that in the previous example that the PowerShell prompt is preceded by [dc01]. This means you're in an interactive PowerShell remoting session to a computer named dc01 and any command you execute is actually run on dc01 and not on your local computer. Also keep in mind that you only have access to the PowerShell commands that exist on the remote computer and not the ones on your local computer while in a remoting session. In other words, if you've installed additional modules on your computer, they won't exist or be accessible on the remote computer.

When you're connected to a remote computer via a one-to-one interactive PowerShell remoting session, you're effectively sitting at the remote computer. The objects are normal objects just like the ones you've been working with throughout this entire book.

```
[dc01]: PS C:\> Get-Process | Get-Member


   TypeName: System.Diagnostics.Process

Name                           MemberType     Definition
----                           ----------     ----------
Handles                        AliasProperty  Handles = Handlecount
Name                           AliasProperty  Name = ProcessName
NPM                            AliasProperty  NPM = NonpagedSystemMemorySize64
PM                             AliasProperty  PM = PagedMemorySize64
SI                             AliasProperty  SI = SessionId
VM                             AliasProperty  VM = VirtualMemorySize64
WS                             AliasProperty  WS = WorkingSet64
Disposed                       Event          System.EventHandler Disposed(System.Object, ...
ErrorDataReceived              Event          System.Diagnostics.DataReceivedEventHandler ...
Exited                         Event          System.EventHandler Exited(System.Object, Sy...
OutputDataReceived             Event          System.Diagnostics.DataReceivedEventHandler ...
BeginErrorReadLine             Method         void BeginErrorReadLine()
BeginOutputReadLine            Method         void BeginOutputReadLine()
CancelErrorRead                Method         void CancelErrorRead()
CancelOutputRead               Method         void CancelOutputRead()
Close                          Method         void Close()
CloseMainWindow                Method         bool CloseMainWindow()
CreateObjRef                   Method         System.Runtime.Remoting.ObjRef CreateObjRef(...
Dispose                        Method         void Dispose(), void IDisposable.Dispose()
Equals                         Method         bool Equals(System.Object obj)
GetHashCode                    Method         int GetHashCode()
GetLifetimeService             Method         System.Object GetLifetimeService()
GetType                        Method         type GetType()
InitializeLifetimeService      Method         System.Object InitializeLifetimeService()
Kill                           Method         void Kill()
Refresh                        Method         void Refresh()
Start                          Method         bool Start()
ToString                       Method         string ToString()
WaitForExit                    Method         bool WaitForExit(int milliseconds), void Wai...
WaitForInputIdle               Method         bool WaitForInputIdle(int milliseconds), boo...
__NounName                     NoteProperty   string __NounName=Process
BasePriority                   Property       int BasePriority {get;}
Container                      Property       System.ComponentModel.IContainer Container {...
EnableRaisingEvents            Property       bool EnableRaisingEvents {get;set;}
ExitCode                       Property       int ExitCode {get;}
ExitTime                       Property       datetime ExitTime {get;}
Handle                         Property       System.IntPtr Handle {get;}
HandleCount                    Property       int HandleCount {get;}
HasExited                      Property       bool HasExited {get;}
Id                             Property       int Id {get;}
MachineName                    Property       string MachineName {get;}
```

```
MainModule                  Property       System.Diagnostics.ProcessModule MainModule ...
MainWindowHandle            Property       System.IntPtr MainWindowHandle {get;}
MainWindowTitle             Property       string MainWindowTitle {get;}
MaxWorkingSet               Property       System.IntPtr MaxWorkingSet {get;set;}
MinWorkingSet               Property       System.IntPtr MinWorkingSet {get;set;}
Modules                     Property       System.Diagnostics.ProcessModuleCollection M...
NonpagedSystemMemorySize    Property       int NonpagedSystemMemorySize {get;}
NonpagedSystemMemorySize64  Property       long NonpagedSystemMemorySize64 {get;}
PagedMemorySize             Property       int PagedMemorySize {get;}
PagedMemorySize64           Property       long PagedMemorySize64 {get;}
PagedSystemMemorySize       Property       int PagedSystemMemorySize {get;}
PagedSystemMemorySize64     Property       long PagedSystemMemorySize64 {get;}
PeakPagedMemorySize         Property       int PeakPagedMemorySize {get;}
PeakPagedMemorySize64       Property       long PeakPagedMemorySize64 {get;}
PeakVirtualMemorySize       Property       int PeakVirtualMemorySize {get;}
PeakVirtualMemorySize64     Property       long PeakVirtualMemorySize64 {get;}
PeakWorkingSet              Property       int PeakWorkingSet {get;}
PeakWorkingSet64            Property       long PeakWorkingSet64 {get;}
PriorityBoostEnabled        Property       bool PriorityBoostEnabled {get;set;}
PriorityClass               Property       System.Diagnostics.ProcessPriorityClass Prio...
PrivateMemorySize           Property       int PrivateMemorySize {get;}
PrivateMemorySize64         Property       long PrivateMemorySize64 {get;}
PrivilegedProcessorTime     Property       timespan PrivilegedProcessorTime {get;}
ProcessName                 Property       string ProcessName {get;}
ProcessorAffinity           Property       System.IntPtr ProcessorAffinity {get;set;}
Responding                  Property       bool Responding {get;}
SafeHandle                  Property       Microsoft.Win32.SafeHandles.SafeProcessHandl...
SessionId                   Property       int SessionId {get;}
Site                        Property       System.ComponentModel.ISite Site {get;set;}
StandardError               Property       System.IO.StreamReader StandardError {get;}
StandardInput               Property       System.IO.StreamWriter StandardInput {get;}
StandardOutput              Property       System.IO.StreamReader StandardOutput {get;}
StartInfo                   Property       System.Diagnostics.ProcessStartInfo StartInf...
StartTime                   Property       datetime StartTime {get;}
SynchronizingObject         Property       System.ComponentModel.ISynchronizeInvoke Syn...
Threads                     Property       System.Diagnostics.ProcessThreadCollection T...
TotalProcessorTime          Property       timespan TotalProcessorTime {get;}
UserProcessorTime           Property       timespan UserProcessorTime {get;}
VirtualMemorySize           Property       int VirtualMemorySize {get;}
VirtualMemorySize64         Property       long VirtualMemorySize64 {get;}
WorkingSet                  Property       int WorkingSet {get;}
WorkingSet64                Property       long WorkingSet64 {get;}
PSConfiguration             PropertySet    PSConfiguration {Name, Id, PriorityClass, Fi...
PSResources                 PropertySet    PSResources {Name, Id, Handlecount, WorkingS...
Company                     ScriptProperty System.Object Company {get=$this.Mainmodule....
CPU                         ScriptProperty System.Object CPU {get=$this.TotalProcessorT...
Description                 ScriptProperty System.Object Description {get=$this.Mainmod...
```

```
FileVersion                 ScriptProperty System.Object FileVersion {get=$this.Mainmod...
Path                        ScriptProperty System.Object Path {get=$this.Mainmodule.Fil...
Product                     ScriptProperty System.Object Product {get=$this.Mainmodule....
ProductVersion              ScriptProperty System.Object ProductVersion {get=$this.Main...

[dc01]: PS C:\>
```

When you're done working with the remote computer, exit the one-to-one remoting session by using the Exit-PSSession cmdlet.

```
[dc01]: PS C:\> Exit-PSSession
```

```
PS C:\>
```

One-To-Many Remoting

Sometimes you may need to perform a task interactively on a remote computer, but remoting is much more powerful when performing a task on multiple remote computers at the same time. The Invoke-Command cmdlet is used to remotely run a command against one or more remote computers at the same time.

```
PS C:\> Invoke-Command -ComputerName dc01, sql02, web01 {Get-Service -Name W32time} -Credentia\
l $Cred
```

```
Status   Name               DisplayName                           PSComputerName
------   ----               -----------                           --------------
Running  W32time            Windows Time                          dc01
Running  W32time            Windows Time                          sql02
Running  W32time            Windows Time                          web01

PS C:\>
```

In the previous example, three servers were queried for the status of the Windows Time service. The Get-Service cmdlet was placed inside the script block of Invoke-Command. Get-Service is actually run on the remote computer and the results are returned to your local computer as deserialized objects.

Piping the previous command to Get-Member shows that the results are indeed deserialized objects.

```
PS C:\> Invoke-Command -ComputerName dc01, sql02, web01 {Get-Service -Name W32time} -Credentia\
l $Cred | Get-Member
```

```
   TypeName: Deserialized.System.ServiceProcess.ServiceController

Name                MemberType   Definition
----                ----------   ----------
GetType             Method       type GetType()
ToString            Method       string ToString(), string ToString(string format, Sys...
Name                NoteProperty string Name=W32time
PSComputerName      NoteProperty string PSComputerName=sql02
PSShowComputerName  NoteProperty bool PSShowComputerName=True
RequiredServices    NoteProperty Deserialized.System.ServiceProcess.ServiceController[...
RunspaceId          NoteProperty guid RunspaceId=570313c4-ac84-4109-bf67-c6b33236af0a
CanPauseAndContinue Property     System.Boolean {get;set;}
CanShutdown         Property     System.Boolean {get;set;}
CanStop             Property     System.Boolean {get;set;}
Container           Property      {get;set;}
DependentServices   Property     Deserialized.System.ServiceProcess.ServiceController[...
DisplayName         Property     System.String {get;set;}
MachineName         Property     System.String {get;set;}
ServiceHandle       Property     System.String {get;set;}
ServiceName         Property     System.String {get;set;}
ServicesDependedOn  Property     Deserialized.System.ServiceProcess.ServiceController[...
ServiceType         Property     System.String {get;set;}
Site                Property      {get;set;}
StartType           Property     System.String {get;set;}
Status              Property     System.String {get;set;}

PS C:\>
```

Also notice that the majority of the methods are missing on deserialized objects which means they're not live objects; They're inert. You can't start or stop a service using a deserialized object because it's a snapshot of the state of that object from the point in time when the command was run on the remote computer.

That doesn't mean you can't start or stop a service using a method with Invoke-Command though. It just means that the method has to be called in the remote session.

I'll stop the Windows Time service on all three of those remote servers using the stop method to prove this point.

```
PS C:\> Invoke-Command -ComputerName dc01, sql02, web01 {(Get-Service -Name W32time).stop(
)} -Credential $Cred
PS C:\> Invoke-Command -ComputerName dc01, sql02, web01 {Get-Service -Name W32time} -Crede
ntial $Cred


Status   Name               DisplayName                           PSComputerName
------   ----               -----------                           --------------
Stopped  W32time            Windows Time                          sql02
Stopped  W32time            Windows Time                          web01
Stopped  W32time            Windows Time                          dc01

PS C:\>
```

As mentioned in a previous chapter, if a cmdlet exists for accomplishing a task, I recommend using it instead of using a method. In the previous scenario, I recommend using the Stop-Service cmdlet instead of the stop method. I chose to use the stop method to prove a point since many people are under the misconception that methods can't be called when using PowerShell remoting. They can't be called on the object that's returned because it's deserialized, but they can be called in the remote session itself.

Windows PowerShell Sessions

In the last example in the previous section, I ran two commands using the Invoke-Command cmdlet. That means two totally separate sessions had to be setup and torn down to run those two commands.

Similarly to how CIM sessions work as discussed in Chapter 7, a PowerShell session can be created to a remote computer so that multiple commands can be run against the remote computer without the overhead of setting up and tearing down a session for each individual command.

Create a PowerShell session to each of the three computers we've been working with in this chapter, DC01, SQL02, and WEB01.

```
PS C:\> $Session = New-PSSession -ComputerName dc01, sql02, web01 -Credential $Cred


PS C:\>
```

Now use the variable named Session that the PowerShell sessions are stored in to start the Windows Time service using a method and then check the status of the service.

```
PS C:\> Invoke-Command -Session $Session {(Get-Service -Name W32time).start()}
PS C:\> Invoke-Command -Session $Session {Get-Service -Name W32time}


Status   Name               DisplayName                           PSComputerName
------   ----               -----------                           --------------
Running  W32time            Windows Time                          web01
Start... W32time            Windows Time                          dc01
Running  W32time            Windows Time                          sql02

PS C:\>
```

Notice that in the previous example, once the session is created using alternate credentials, it's no longer necessary to specify the credentials each time a command is run.

When you're finished using the sessions, be sure to remove them.

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


PS C:\>
```

Summary

In this chapter you've learned about PowerShell remoting and how to run commands in an interactive session against one remote computer and how to run commands against multiple computers using one-to-many remoting. You've also learned the benefits of using a PowerShell session when running multiple commands against the same remote computer.

Review

1. How do you enable PowerShell remoting?
2. What is the PowerShell command for starting an interactive session with a remote computer?
3. What is one of the benefits of using a PowerShell remoting session versus just specifying the computer name with each command?
4. Can a PowerShell remoting session be used with a one-to-one remoting session?
5. What is the difference in the type of objects that are returned by cmdlets versus those returned when running those same cmdlets against remote computers with Invoke-Command?

Recommended Reading

- about_Remote
- about_Remote_FAQ
- about_Remote_Output
- about_Remote_Requirements
- about_Remote_Troubleshooting
- about_Remote_Variables

Chapter 9 - Functions

If you're writing PowerShell one-liners or scripts and find yourself often having to modify them for different scenarios, or if you're sharing the code with others, there's a good chance that particular one-liner or script is a good candidate to be turned into a function so that it can be used as a reusable tool.

Whenever possible, I prefer to write a function because to me it's more tool oriented. I can place the function in a script module, place that module in the $env:PSModulePath and with PowerShell version 3.0 or higher, I can simply call the function without needing to physically locate where it's saved. With PowerShellGet which was introduced in PowerShell version 5.0, it's also easier to share those modules in a NuGet repository. PowerShellGet is available as a separate download for PowerShell version 3.0 and higher.

Don't overcomplicate things. Keep it simple and use the most straight forward way to accomplish a task. Avoid aliases and positional parameters in scripts and functions and any code that you share. Format your code for readability. Don't hard code values (don't use static values), use parameters and variables. Don't write unnecessary code even if it doesn't hurt anything because it adds unnecessary complexity. Attention to detail goes a long way when writing any PowerShell code.

Naming

When naming your functions in PowerShell, use a pascal case name with an approved verb and a singular noun. I also recommend prefixing the noun. For example: ApprovedVerb-PrefixSingularNoun.

In PowerShell, there's a specific list of approved verbs which can be obtained by running Get-Verb.

```
PS C:\> Get-Verb | Sort-Object -Property Verb


Verb        Group
----        -----
Add         Common
Approve     Lifecycle
Assert      Lifecycle
Backup      Data
Block       Security
Checkpoint  Data
Clear       Common
Close       Common
Compare     Data
```

```
Complete    Lifecycle
Compress    Data
Confirm     Lifecycle
Connect     Communications
Convert     Data
ConvertFrom Data
ConvertTo   Data
Copy        Common
Debug       Diagnostic
Deny        Lifecycle
Disable     Lifecycle
Disconnect  Communications
Dismount    Data
Edit        Data
Enable      Lifecycle
Enter       Common
Exit        Common
Expand      Data
Export      Data
Find        Common
Format      Common
Get         Common
Grant       Security
Group       Data
Hide        Common
Import      Data
Initialize  Data
Install     Lifecycle
Invoke      Lifecycle
Join        Common
Limit       Data
Lock        Common
Measure     Diagnostic
Merge       Data
Mount       Data
Move        Common
New         Common
Open        Common
Optimize    Common
Out         Data
Ping        Diagnostic
Pop         Common
Protect     Security
Publish     Data
Push        Common
Read        Communications
Receive     Communications
```

```
Redo        Common
Register    Lifecycle
Remove      Common
Rename      Common
Repair      Diagnostic
Request     Lifecycle
Reset       Common
Resize      Common
Resolve     Diagnostic
Restart     Lifecycle
Restore     Data
Resume      Lifecycle
Revoke      Security
Save        Data
Search      Common
Select      Common
Send        Communications
Set         Common
Show        Common
Skip        Common
Split       Common
Start       Lifecycle
Step        Common
Stop        Lifecycle
Submit      Lifecycle
Suspend     Lifecycle
Switch      Common
Sync        Data
Test        Diagnostic
Trace       Diagnostic
Unblock     Security
Undo        Common
Uninstall   Lifecycle
Unlock      Common
Unprotect   Security
Unpublish   Data
Unregister  Lifecycle
Update      Data
Use         Other
Wait        Lifecycle
Watch       Common
Write       Communications

PS C:\>
```

In the previous example, I've sorted the results by the Verb column. The Group column gives you an idea of how these verbs are used. The reason it's important to choose an approved verb in PowerShell is once the functions are added to a module, the module will generate a warning message if you choose an unapproved verb. That warning message will make your functions look unprofessional. Unapproved verbs also limit the discoverability of your functions.

A Simple function

A function in PowerShell is declared with the function keyword followed by the function name and then an open and closing curly brace. The code that the function will execute is contained within those curly braces.

```
function Get-Version {
    $PSVersionTable.PSVersion
}
```

The PowerShell function shown in the previous example is a very simple example. It returns the version of PowerShell.

```
PS C:\> Get-Version


Major  Minor  Build  Revision
-----  -----  -----  --------
5      1      14393  693

PS C:\>
```

There’s a good chance of name conflict with functions named something like Get-Version and default commands in PowerShell or commands that others may write. This is why I recommend prefixing the noun portion of your functions to help prevent naming conflicts. In the following example, I’ll use the prefix “PS”.

```
function Get-PSVersion {
    $PSVersionTable.PSVersion
}
```

Other than the name, this function is identical to the previous one.

```
PS C:\> Get-PSVersion
```

```
Major  Minor  Build  Revision
-----  -----  -----  --------
5      1      14393  693

PS C:\>
```

Even when prefixing the noun with something like PS, there's still a good chance of having a name conflict. I typically prefix my function nouns with my initials. Develop a standard and stick to it.

```
function Get-MrPSVersion {
    $PSVersionTable.PSVersion
}
```

This function is no different than the previous two other than using a more sensible name to try to prevent naming conflicts with other PowerShell commands.

```
PS C:\> Get-MrPSVersion


Major  Minor  Build  Revision
-----  -----  -----  --------
5      1      14393  693

PS C:\>
```

Once loaded into memory, you can see functions on the Function PSDrive.

```
PS C:\> Get-ChildItem -Path Function:\Get-*Version


CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Get-Version
Function        Get-PSVersion
Function        Get-MrPSVersion

PS C:\>
```

If you want to remove these functions from your current session, you'll have to remove them from the Function PSDrive or close and re-open PowerShell.

```
PS C:\> Get-ChildItem -Path Function:\Get-*Version | Remove-Item
PS C:\>
```

Verify that the functions were indeed removed.

```
PS C:\> Get-ChildItem -Path Function:\Get-*Version


PS C:\>
```

If the functions were loaded as part of a module, the module can simply be unloaded to remove them.

```
PS C:\> Remove-Module -Name <ModuleName>
```

The Remove-Module cmdlet removes modules from memory in your current PowerShell session, it doesn't remove them from your system or from disk.

Parameters

Don't statically assign values! Use parameters and variables. When it comes to naming your parameters, use the same name as the default cmdlets for your parameter names whenever possible.

```
function Test-MrParameter {

    param (
        $ComputerName
    )

    Write-Output $ComputerName

}
```

Why did I use ComputerName and not Computer, ServerName, or Host for my parameter name? It's because I wanted my function standardized like the default cmdlets.

I'll create a function to query all of the commands on a system and return the number of them that have specific parameter names.

```
function Get-MrParameterCount {
    param (
        [string[]]$ParameterName
    )

    foreach ($Parameter in $ParameterName) {
        $Results = Get-Command -ParameterName $Parameter -ErrorAction SilentlyContinue

        [pscustomobject]@{
            ParameterName = $Parameter
            NumberOfCmdlets = $Results.Count
        }
    }
}
```

As you can see in the results shown below, there are 39 commands that have a ComputerName parameter and there aren't any that have parameters such as Computer, ServerName, Host, or Machine.

```
PS C:\> Get-MrParameterCount -ParameterName ComputerName, Computer,
ServerName, Host, Machine
```

```
ParameterName NumberOfCmdlets
------------- ---------------
ComputerName               39
Computer                    0
ServerName                  0
Host                        0
Machine                     0

PS C:\>
```

I also recommend using the same case for your parameter names as the default cmdlets. Use ComputerName, not computername. This will make your functions look and feel like the default cmdlets which means that people who are already familiar with PowerShell will feel right at home.

Advanced Functions

Turning a function in PowerShell into an advanced function is really simple. One of the differences in a function and an advanced function is that advanced functions have a number of common parameters that are added to the function automatically. These common parameters include parameters such as Verbose and Debug.

I'll start out with the Test-MrParameter function that was used in the previous section.

```
function Test-MrParameter {

    param (
        $ComputerName
    )

    Write-Output $ComputerName

}
```

What I want you to notice is that the Test-MrParameter function doesn't have any common parameters. There are a couple of different ways to see the common parameters. One is by viewing the syntax using Get-Command.

```
PS C:\> Get-Command -Name Test-MrParameter -Syntax


Test-MrParameter [[-ComputerName] <Object>]

PS C:\>
```

Another is to drill down into the parameters with Get-Command.

```
PS C:\> (Get-Command -Name Test-MrParameter).Parameters.Keys


ComputerName
PS C:\>
```

Add CmdletBinding to turn the function into an advanced function.

```
function Test-MrCmdletBinding {

    [CmdletBinding()] #<<-- This turns a regular function into an advanced function
    param (
        $ComputerName
    )

    Write-Output $ComputerName

}
```

Simply adding CmdletBinding adds the common parameters automatically. CmdletBinding does require a param block, but the param block can be empty.

```
PS C:\> Get-Command -Name Test-MrCmdletBinding -Syntax


Test-MrCmdletBinding [[-ComputerName] <Object>] [<CommonParameters>]

PS C:\>
```

Drilling down into the parameters with Get-Command shows the actual parameter names including the common ones.

```
PS C:\> (Get-Command -Name Test-MrCmdletBinding).Parameters.Keys


ComputerName
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
InformationVariable
OutVariable
OutBuffer
PipelineVariable
PS C:\>
```

SupportsShouldProcess

SupportsShouldProcess adds WhatIf and Confirm parameters. This is only needed for commands that make changes.

```
function Test-MrSupportsShouldProcess {

    [CmdletBinding(SupportsShouldProcess)]
    param (
        $ComputerName
    )

    Write-Output $ComputerName

}
```

Notice that there are now WhatIf and Confirm parameters.

```
PS C:\> Get-Command -Name Test-MrSupportsShouldProcess -Syntax


Test-MrSupportsShouldProcess [[-ComputerName] <Object>] [-WhatIf] [-Confirm] [<CommonParameter\
s>]

PS C:\>
```

Once again, you can also use Get-Command to return a list of the actual parameter names including the common ones along with WhatIf and Confirm.

```
PS C:\> (Get-Command -Name Test-MrSupportsShouldProcess).Parameters.Keys


ComputerName
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
InformationVariable
OutVariable
OutBuffer
PipelineVariable
WhatIf
Confirm
PS C:\>
```

Parameter Validation

Validate input early on. Why allow your code to continue on a path when it's not possible to successfully complete without valid input?

Always type the variables that are being used for your parameters (specify a datatype).

```
function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [string]$ComputerName
    )

    Write-Output $ComputerName

}
```

In the previous example, I've specified String as the datatype for the ComputerName parameter. This causes it to allow only a single computer name to be specified. If more than one computer name is specified via a comma separated list, an error will be generated.

```
PS C:\> Test-MrParameterValidation -ComputerName Server01, Server02


Test-MrParameterValidation : Cannot process argument transformation on parameter
'ComputerName'. Cannot convert value to type System.String.
At line:1 char:42
+ Test-MrParameterValidation -ComputerName Server01, Server02
+
    + CategoryInfo          : InvalidData: (:) [Test-MrParameterValidation], ParameterBi
   ndingArgumentTransformationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,Test-MrParameterValid
   ation

PS C:\>
```

The problem at this point is that it's perfectly valid to not specify a computer name at all and at least one computer name needs to be specified otherwise the function can't possibly complete successfully. This is where the Mandatory parameter attribute comes in handy.

```
function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [string]$ComputerName
    )

    Write-Output $ComputerName

}
```

The syntax used in the previous example is PowerShell version 3.0 and higher compatible. [Parameter(Mandatory=$true)] could be specified instead to make the function compatible with PowerShell version 2.0 and higher. Now that the ComputerName is required, if one isn't specified, the function will prompt for one.

```
PS C:\> Test-MrParameterValidation


cmdlet Test-MrParameterValidation at command pipeline position 1
Supply values for the following parameters:
ComputerName:
```

If you want to allow for more than one value to be specified for the ComputerName parameter, use the String datatype but add open and closed square brackets to the datatype to allow for an array of strings.

```
function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [string[]]$ComputerName
    )

    Write-Output $ComputerName

}
```

Maybe you want to specify a default value for the ComputerName parameter if one is not specified. The problem is that default values cannot be used with mandatory parameters. Instead, you'll need to use the ValidateNotNullOrEmpty parameter validation attribute with a default value.

```
function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    Write-Output $ComputerName

}
```

Even when setting a default value, try not to use static values. In the previous example, $env:COMPUTERNAME is used as the default value which will automatically be translated into the local computer name if a value is not provided.

Verbose Output

While inline comments are useful, especially if you writing some unusual code that may not be straight forward, they'll never been seen by someone who is using your functions unless they dig into the code itself.

The function shown in the following example has an inline comment in the foreach loop. While this particular comment may not be that difficult to locate, imagine if the function included hundreds of lines of code.

```
function Test-MrVerboseOutput {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    foreach ($Computer in $ComputerName) {
        #Attempting to perform some action on $Computer <<-- Don't use
        #inline comments like this, use write verbose instead.
        Write-Output $Computer
    }

}
```

A better option is to use Write-Verbose instead of inline comments.

```
function Test-MrVerboseOutput {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    foreach ($Computer in $ComputerName) {
        Write-Verbose -Message "Attempting to perform some action on $Computer"
        Write-Output $Computer
    }

}
```

When the function is called without the Verbose parameter, the verbose output won't be displayed.

```
Test-MrVerboseOutput -ComputerName Server01, Server02
```

When it’s called with the Verbose parameter, the verbose output will be displayed.

```
Test-MrVerboseOutput -ComputerName Server01, Server02 -Verbose
```

Pipeline Input

When you want your function to accept pipeline input, some additional coding is necessary. As mentioned earlier in this book, commands can accept pipeline input by value (by type) or by property name. You can write your functions just like the native commands so they can accept either one or both of these types of input.

To accept pipeline input by value, specified the ValueFromPipeline parameter attribute for that particular parameter. Keep in mind that you can only accept pipeline input by value from one of each datatype. For example, if you have two parameters that accept string input, only one of those can accept pipeline input by value because if you specified it for both of the string parameters, the pipeline input wouldn’t know which one to bind to. This is another reason I call this type of pipeline input by type instead of by value.

Pipeline input comes in one item at a time similar to the way a foreach loop works. At a minimum, a PROCESS block is required to process each of these items if you’re accepting an array as input. If you’re only accepting a single value as input, a process block isn’t necessary, but I still recommend going ahead and specifying it for consistency.

```
function Test-MrPipelineInput {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline)]
        [string[]]$ComputerName
    )

    PROCESS {
        Write-Output $ComputerName
    }

}
```

Accepting pipeline input by property name is similar except it’s specified with the ValueFromPipelineByPropertyName parameter attribute and it can be specified for any number of parameters regardless of datatype. The key is that the output of the command that’s being piped in has to have a property name that matches the name of the parameter or a parameter alias of your function.

```
function Test-MrPipelineInput {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
            Write-Output $ComputerName
    }

}
```

BEGIN and END blocks are optional. BEGIN would be specified before the PROCESS block and is used to perform any initial work prior to the items being received from the pipeline. This is very important to understand. Values that are piped in are NOT accessible in the BEGIN block. The END block would be specified after the PROCESS block and is used for cleanup once all of the items that are piped in have been processed.

Error Handling

The function shown in the following example will generate an unhandled exception if a computer cannot be contacted.

```
function Test-MrErrorHandling {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($Computer in $ComputerName) {
            Test-WSMan -ComputerName $Computer
        }
    }

}
```

There are a couple of different ways to handle errors in PowerShell. Try / Catch is the more modern way to handle errors.

```
function Test-MrErrorHandling {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Test-WSMan -ComputerName $Computer
            }
            catch {
                Write-Warning -Message "Unable to connect to Computer: $Computer"
            }
        }
    }

}
```

Although the function shown in the previous example uses error handling, it also generates an unhandled exception because the command doesn't generate a terminating error. This is also very important to understand. Only terminating errors are caught. Specify the ErrorAction parameter with Stop as the value to turn a non-terminating error into a terminating one.

```
function Test-MrErrorHandling {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Test-WSMan -ComputerName $Computer -ErrorAction Stop
            }
            catch {
                Write-Warning -Message "Unable to connect to Computer: $Computer"
            }
        }
    }

}
```

Don't modify the global $ErrorActionPreference variable unless absolutely necessary. If you're using something like the .NET Framework directly from within your PowerShell function, you won't be able to specify the ErrorAction on the command itself. In that scenario, you might need to change the global $ErrorActionPreference variable, but if you do change it, change it back immediately after trying the command.

Comment-Based Help

It's considered to be a best practice to add comment based help to your functions so the people you're sharing them with will know how to use them.

```
function Get-MrAutoStoppedService {

<#
.SYNOPSIS
    Returns a list of services that are set to start automatically, are not
    currently running, excluding the services that are set to delayed start.

.DESCRIPTION
    Get-MrAutoStoppedService is a function that returns a list of services from
    the specified remote computer(s) that are set to start automatically, are not
    currently running, and it excludes the services that are set to start automatically
    with a delayed startup.

.PARAMETER ComputerName
    The remote computer(s) to check the status of the services on.

.PARAMETER Credential
    Specifies a user account that has permission to perform this action. The default
    is the current user.

.EXAMPLE
     Get-MrAutoStoppedService -ComputerName 'Server1', 'Server2'

.EXAMPLE
     'Server1', 'Server2' | Get-MrAutoStoppedService

.EXAMPLE
     Get-MrAutoStoppedService -ComputerName 'Server1' -Credential (Get-Credential)

.INPUTS
```

```
    String

.OUTPUTS
    PSCustomObject

.NOTES
    Author:  Mike F Robbins
    Website: http://mikefrobbins.com
    Twitter: @mikefrobbins
#>

    [CmdletBinding()]
    param (

    )

    #Function Body

}
```

When you add comment based help to your functions, help can be retrieved for them just like the default built-in commands.

All of the syntax for writing a function in PowerShell can seem overwhelming especially for someone who is just getting started. Often times if I can't remember the syntax for something, I'll open a second copy of the ISE on a separate monitor and view the "Cmdlet (advanced function) - Complete" snippet while typing in the code for my function. Snippets can be access in the PowerShell ISE using the Cntl + J key combination.

Summary

In this chapter you've learned the basics of writing functions in PowerShell to include how to turn a function into an advanced function and some of the more important elements that you should consider when writing PowerShell functions such as parameter validation, verbose output, pipeline input, error handling, and comment based help.

Review

1. How do you obtain a list of approved verbs in PowerShell?
2. How do you turn a PowerShell function into an advanced function?
3. When should WhatIf and Confirm parameters be added to your PowerShell functions?
4. How do you turn a non-terminating error into a terminating one?
5. Why should you add comment based help to your functions?

Recommended Reading

- about_Functions
- about_Functions_Advanced_Parameters
- about_CommonParameters
- about_Functions_CmdletBindingAttribute
- about_Functions_Advanced
- about_Try_Catch_Finally
- about_Comment_Based_Help
- Video: PowerShell Toolmaking with Advanced Functions and Script Modules

Chapter 10 - Script Modules

Turning your one-liners and scripts in PowerShell into reusable tools in the form of creating functions as shown in the previous chapter, is what you want to strive for. It becomes even more important if it's something that you're going to use frequently. Once you have your functions created, you'll want to package them up as script modules to make them look and feel more professional, as well as make them easier to share.

Dot-Sourcing Functions

Something that we didn't talk about in the previous chapter is dot-sourcing functions. When a function isn't part of a module, the only way to load it into memory short of opening up the ISE and running it, is to dot-source the PS1 file that it's saved in.

The following function has been saved as Get-MrPSVersion.ps1.

```
function Get-MrPSVersion {
    $PSVersionTable
}
```

If you simply run the script, nothing happens.

```
.\Get-MrPSVersion.ps1
```

If you try to call the function, it generates an error message.

```
PS C:\> Get-MrPSVersion


Get-MrPSVersion : The term 'Get-MrPSVersion' is not recognized as the name of a cmdlet,
function, script file, or operable program. Check the spelling of the name, or if a path
was included, verify that the path is correct and try again.
At line:1 char:1
+ Get-MrPSVersion
    + CategoryInfo          : ObjectNotFound: (Get-MrPSVersion:String) [], CommandNotFou
   ndException
    + FullyQualifiedErrorId : CommandNotFoundException
PS C:\>
```

You can determine if functions are loaded into memory by checking to see if they exist on the function PSDrive.

```
PS C:\> Get-ChildItem -Path Function:\Get-MrPSVersion
```

```
Get-ChildItem : Cannot find path 'Get-MrPSVersion' because it does not exist.
At line:1 char:1
+ Get-ChildItem -Path Function:\Get-MrPSVersion
    + CategoryInfo          : ObjectNotFound: (Get-MrPSVersion:String) [Get-ChildItem],
   ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCom
   mand

PS C:\>
```

As shown in the previous set of results, it’s not found on the function PSDrive.

The problem with simply calling the script that contains the function is that the functions are loaded in the Script scope and when the script completes, that scope is removed and the function is removed with it.

The function needs to be loaded into the Global scope and that can be accomplished by dot-sourcing the script that contains the function. The relative path can be used.

```
. .\Get-MrPSVersion.ps1
```

The fully qualified path can also be used.

```
. C:\Demo\Get-MrPSVersion.ps1
```

If a portion of the path is stored in a variable, it can be combined with the remainder of the path. There’s no reason to use string concatenation to combine the variable together with the remainder of the path.

```
$Path = 'C:\'
. $Path\Get-MrPSVersion.ps1
```

Now when the function PSDrive in PowerShell is checked, the Get-MrPSVersion function does exist.

```
PS C:\> Get-ChildItem -Path Function:\Get-MrPSVersion
```

```
CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Get-MrPSVersion

PS C:\>
```

Script Modules

A script module in PowerShell is simply a file containing one or more functions that's saved as a PSM1 file instead of a PS1 file.

How do you create a script module? You're probably guessing with a command named something like New-Module. Your assumption would be wrong. While there is a command in PowerShell named New-Module, that command creates a dynamic module, not a script module. Always be sure to read the help for a command even when you think you've found the command you need.

```
PS C:\> help New-Module


NAME
    New-Module

SYNOPSIS
    Creates a new dynamic module that exists only in memory.

SYNTAX
    New-Module [-Name] <String> [-ScriptBlock] <ScriptBlock> [-ArgumentList <Object[]>]
    [-AsCustomObject] [-Cmdlet <String[]>] [-Function <String[]>] [-ReturnResult]
    [<CommonParameters>]

DESCRIPTION
    The New-Module cmdlet creates a dynamic module from a script block. The members of
    the dynamic module, such as functions and variables, are immediately available in
    the session and remain available until you close the session.

    Like static modules, by default, the cmdlets and functions in a dynamic module are
    exported and the variables and aliases are not. However, you can use the
    Export-ModuleMember cmdlet and the parameters of New-Module to override the defaults.

    You can also use the AsCustomObject parameter of New-Module to return the dynamic
    module as a custom object. The members of the modules, such as functions, are
    implemented as script methods of the custom object instead of being imported into
    the session.

    Dynamic modules exist only in memory, not on disk. Like all modules, the members of
    dynamic modules run in a private module scope that is a child of the global scope.
    Get-Module cannot get a dynamic module, but Get-Command can get the exported members.

    To make a dynamic module available to Get-Module , pipe a New-Module command to
    Import-Module, or pipe the module object that New-Module returns to Import-Module .
    This action adds the dynamic module to the Get-Module list, but it does not save the
    module to disk or make it persistent.

RELATED LINKS
    Online Version: http://go.microsoft.com/fwlink/?LinkId=821495
    Export-ModuleMember
    Get-Module
    Import-Module
    Remove-Module

REMARKS
    To see the examples, type: "get-help New-Module -examples".
    For more information, type: "get-help New-Module -detailed".
    For technical information, type: "get-help New-Module -full".
    For online help, type: "get-help New-Module -online"

PS C:\>
```

In the previous chapter I mentioned that functions should use approved verbs otherwise they'll generate a warning message when the module is imported. The following code, which uses the New-Module cmdlet to create a dynamic module in memory, is used to demonstrate the unapproved verb warning.

```
New-Module -Name MyModule -ScriptBlock {

    function Return-MrOsVersion {
        Get-CimInstance -ClassName Win32_OperatingSystem |
        Select-Object -Property @{label='OperatingSystem';expression={$_.Caption}}
    }

    Export-ModuleMember -Function Return-MrOsVersion

} | Import-Module
```

```
WARNING: The names of some imported commands from the module 'MyModule' include
unapproved verbs that might make them less discoverable. To find the commands with
unapproved verbs, run the Import-Module command again with the Verbose parameter. For a
list of approved verbs, type Get-Verb.
PS C:\>
```

Just to reiterate, although the New-Module cmdlet was used in the previous example, that's not the command for creating script modules in PowerShell.

Save the following two functions in a file named MyScriptModule.psm1.

```
function Get-MrPSVersion {
    $PSVersionTable
}

function Get-MrComputerName {
    $env:COMPUTERNAME
}
```

Try to call one of the functions.

```
PS C:\> Get-MrComputerName
```

```
Get-MrComputerName : The term 'Get-MrComputerName' is not recognized as the name of a
cmdlet, function, script file, or operable program. Check the spelling of the name, or
if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ Get-MrComputerName
    + CategoryInfo          : ObjectNotFound: (Get-MrComputerName:String) [], CommandNot
   FoundException
    + FullyQualifiedErrorId : CommandNotFoundException

PS C:\>
```

An error message is generated saying the function can't be found. You could also check the function PSDrive just like before and you'll find that it doesn't exist there either.

You could manually import the file with the Import-Module cmdlet.

```
Import-Module C:\MyScriptModule.psm1
```

Module autoloading is a feature that was introduced in PowerShell version 3. In order to take advantage of module autoloading, a script module needs to be saved in a folder with the same base name as the PSM1 file and in a location specified in $env:PSModulePath.

```
PS C:\> $env:PSModulePath


C:\Users\mike-ladm\Documents\WindowsPowerShell\Modules;C:\Program Files\WindowsPowerShell\
Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules;C:\Program Files (x86)\Microsof
t SQL Server\130\Tools\PowerShell\Modules\
PS C:\>
```

The results are kind of difficult to read. Since the paths are separated by a semicolon, you can split the results on those semicolons to return each path on a separate line. This will make them easier to read.

```
PS C:\> $env:PSModulePath -split ';'


C:\Users\mike-ladm\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules
C:\Program Files (x86)\Microsoft SQL Server\130\Tools\PowerShell\Modules\
PS C:\>
```

The first three items in the previous list are the default ones and when SQL Server Management Studio was installed, it added the last one. In order for module autoloading to work, the MyScript-Module.psm1 file would need to be located in a folder named MyScriptModule directly inside one of those paths.

Not so fast. For me, the current user path is the first one in the list. I almost never use that path since I log into Windows as a different user in which I run PowerShell. That means it's not located in the Documents folder of the user in which I'm logged into Windows.

The second path was added in PowerShell version 4.0 and is the AllUsers path. This is the location where I store all of my modules.

The third path resides underneath Windows\System32. Based on information that I received from the PowerShell team at Microsoft, only Microsoft should be storing modules in that location since it resides underneath the operating systems folder.

Once the PSM1 file is located in the correct path, the module will load automatically when one of its commands is called, as long as you're running PowerShell version 3.0 or higher.

Module Manifests

All modules should have a module manifest. A module manifest contains metadata about your module. The file extension for a module manifest file is PSD1. Not all files with a PSD1 extension are module manifests. They can also be used for things such as storing the environmental portion of a DSC (Desired State Configuration) configuration. New-ModuleManifest is used to create a module manifest. Path is the only value that's required. However, the module won't work if root module isn't specified. It's a good idea to specify Author and Description in case you decide to upload your module to a Nuget repository with PowerShellGet since those values are required in that scenario.

The version of a module without a manifest is 0.0 (This is a dead giveaway that the module doesn't have a manifest).

```
PS C:\> Get-Module -Name MyScriptModule


ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     0.0        myscriptmodule                      {Get-MrComputerName, Get-MrP...

PS C:\>
```

The module manifest can be created with all of the recommended information.

```
PS C:\> New-ModuleManifest -Path $env:ProgramFiles\WindowsPowerShell\Modules\MyScriptModul
e\MyScriptModule.psd1 -RootModule MyScriptModule -Author 'Mike F Robbins' -Description 'My
ScriptModule' -CompanyName 'mikefrobbins.com'
PS C:\>
```

If any of this information is missed during the initial creation of the module manifest, it can be added or updated later using Update-ModuleManifest. Don't recreate the manifest using New-ModuleManifest once it's already created because the GUID will change.

Defining Public and Private Functions

You may have helper functions that you may want to get private that are only accessible by other functions within the module and not accessible to be called directly by the users of your module. There are a couple of different ways to accomplish this.

If you're not following the best practices and only have a PSM1 file then your only option is to use the Export-ModuleMember cmdlet.

```
function Get-MrPSVersion {
    $PSVersionTable
}

function Get-MrComputerName {
    $env:COMPUTERNAME
}

Export-ModuleMember -Function Get-MrPSVersion
```

In the previous example, only the Get-MrPSVersion function will be available to the users of your module, but the Get-MrComputerName function will be available to other functions within the module itself.

```
PS C:\> Get-Command -Module MyScriptModule

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Get-MrPSVersion                                    1.0        MyScript...

PS C:\>
```

If you've added a module manifest to your module (and you should), then I recommend specifying the individual functions you want to export in the FunctionsToExport section of the module manifest.

```
FunctionsToExport = 'Get-MrPSVersion'
```

It's not necessary to use both Export-ModuleMember in the PSM1 file and the FunctionsToExport section of the module manifest. One or the other is sufficient.

Summary

In this chapter you've learned how to turn your functions into a script module in PowerShell. You've also leaned some of the best practices for creating script modules such as creating a module manifest for your script module.

Review

1. How do you create a script module in PowerShell?
2. Why is it important for your functions to use an approved verb?
3. How do you create a module manifest in PowerShell?
4. What are the two options for exporting only certain functions from your module?
5. What is required for your modules to load automatically when a command is called?

Recommended Reading

- How to Create PowerShell Script Modules and Module Manifests
- about_Modules
- New-ModuleManifest
- Export-ModuleMember

Appendix A - The Cryptic Help Syntax

View the syntax section of the help for a command such as in the following example where help is retrieved for the Get-EventLog cmdlet.

```
PS C:\> help Get-EventLog



NAME
    Get-EventLog

SYNOPSIS
    Gets the events in an event log, or a list of the event logs, on the local or remote
    computers.


SYNTAX
    Get-EventLog [-LogName] <String> [[-InstanceId] <Int64[]>] [-After <DateTime>]
    [-AsBaseObject] [-Before <DateTime>] [-ComputerName <String[]>] [-EntryType {Error |
    Information | FailureAudit | SuccessAudit | Warning}] [-Index <Int32[]>] [-Message
    <String>] [-Newest <Int32>] [-Source <String[]>] [-UserName <String[]>]
    [<CommonParameters>]

    Get-EventLog [-AsString] [-ComputerName <String[]>] [-List] [<CommonParameters>]
```

Only the relevant portion of the help is shown in the previous example.

The cryptic syntax is primarily made up of several sets of opening "[" and closing "]" square brackets. These have two different meanings depending on how they're used. Anything contained within square brackets is optional unless they're a set of empty "[]" square brackets. Empty square brackets are only ever shown after a datatype such as <string[]> which means that particular parameter can accept more than one value.

The first parameter in the first parameter set of Get-EventLog is LogName. LogName is surrounded by square brackets which means that it's a positional parameter. In other words, specifying the name of the parameter itself is optional as long as it's specified in the correct position. Based on the information provided in the angle brackets after the parameter name, it needs a single string value. The entire parameter name and datatype are not surrounded by square brackets so the LogName parameter is required when using this parameter set.

```
Get-EventLog [-LogName] <String>
```

The second parameter is InstanceId. Notice that the parameter name and the datatype are both completely surrounded by square brackets. This means the InstanceId parameter is optional (not mandatory). Also notice that InstanceId is surrounded by its own set of square brackets. As with the LogName parameter, this means the parameter is positional. There's one last set of square brackets after the datatype. As previously mentioned, this means that it can accept more than one value in the form of an array or a comma separated list.

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

The second parameter set has a List parameter. It's a switch parameter because there's no datatype following the parameter name. When the List parameter is specified, it's on and when it's not specified, it's off.

```
[-List]
```

The syntax information for a command can also be retrieved using Get-Command by specifying the Syntax parameter. This is a handy shortcut that I use all the time. It allows me to quickly learn how to use a command without having to sift through multiple pages of help information. If I end up needing more information, then I'll revert to using the actual help content.

```
PS C:\> Get-Command -Name Get-EventLog -Syntax
```

```
Get-EventLog [-LogName] <string> [[-InstanceId] <long[]>] [-ComputerName <string[]>] [-New
est <int>] [-After <datetime>] [-Before <datetime>] [-UserName <string[]>] [-Index <int[]>
] [-EntryType <string[]>] [-Source <string[]>] [-Message <string>] [-AsBaseObject] [<Commo
nParameters>]

Get-EventLog [-ComputerName <string[]>] [-List] [-AsString] [<CommonParameters>]

PS C:\>
```

The more you use the help system in PowerShell, the easier remembering all of the different nuances will become and before you know it, using it will be second nature.

Appendix B - Other Resources

Congratulations, you've successfully completed this book! Learning PowerShell is, however, a never ending journey. This appendix provides a list of resources to assist you in continuing your journey from this point forward.

With the exception of the URL for blogs, the following links reference external websites the author has no control over so proceed to these sites with caution and at your own risk. Some are specific to PowerShell, and others to technologies such as SQL Server. Be sure to read and follow all guidelines on the referenced sites before posting any information to them.

- PowerShell Documentation
- PowerShell Best Practices and Style Guide
- Microsoft Virtual Academy
- PowerShell.org
- GitHub
- User Groups
- Twitter
- Slack
- Blogs
- PowerShell Saturday
- PowerShell Virtual Chapter of SQL PASS
- SQL Saturday
- Trello

The authors blog site is referenced as a location where to find his and other blog sites. Specifically, the blogroll section of his website contains a list of other blogs that he recommends.