

**Michael Shepard, Chendrayan Venkatesan,
Sherif Talaat, Brenton J.W. Blawat**

PowerShell: Automating Administrative Tasks

Learning Path

Learn PowerShell from the inside out, right from basic scripting all the way to becoming a master at automating, managing, and maintaining your Windows environment



Packt

PowerShell: Automating Administrative Tasks

Table of Contents

[PowerShell: Automating Administrative Tasks](#)

[PowerShell: Automating Administrative Tasks](#)

[Credits](#)

[Preface](#)

[What this learning path covers](#)

[What you need for this learning path](#)

[Who this learning path is for](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Module 1](#)

[1. First Steps](#)

[Determining the installed PowerShell version](#)

[Using the registry to find the installed version](#)

[Using PowerShell to find the installed version](#)

[Installing/upgrading PowerShell](#)

[Starting a PowerShell session](#)

[PowerShell hosts](#)

[64-bit and 32-bit PowerShell](#)

[PowerShell as an administrator](#)

[Simple PowerShell commands](#)

[PowerShell aliases](#)

[Summary](#)

[For further reading](#)

[2. Building Blocks](#)

[What can you do?](#)

[The scripter's secret weapon – tab completion](#)

[How does that work?](#)

[Interpreting the cmdlet syntax](#)

[Summary](#)

[For further reading](#)

[3. Objects and PowerShell](#)

[Objects all the way down](#)

[Digging into objects](#)

[Types, classes, and objects](#)

[What are members?](#)

[The DOS DIR command](#)

[The IPCONFIG command](#)

[PowerShell for comparison](#)

[The Get-Member cmdlet](#)

[Where did these all come from?](#)

[Summary](#)

[For further reading](#)

[4. Life on the Assembly Line](#)

[The pipeline as an assembly line](#)

[This isn't your DOS or Linux pipeline](#)

[Objects at your disposal](#)

[Dealing with pipeline data](#)

[The Sort-Object cmdlet](#)

[The Where-Object cmdlet](#)

[The Select-Object cmdlet](#)

[Limiting the number of objects returned](#)

[Limiting the properties of the objects returned](#)

[Retrieving the values of a single property](#)

[The Measure-Object cmdlet](#)

[The Group-Object cmdlet](#)

[Putting them together](#)

[Summary](#)

[For further reading](#)

[5. Formatting Output](#)

[When does formatting occur?](#)

[The rules of automatic formatting](#)

[Formatting files](#)

[Formatting decisions are based on the first object](#)

[Small objects go in a table](#)

[Large objects go in a list](#)

[Cmdlets that control formatting](#)

[Format-Table and Format-List](#)

[The dangers of formatting](#)

[Best practices of formatting](#)

[Summary](#)

[For further reading](#)

[6. Scripts](#)

[Packaging commands](#)

[Execution policy](#)

[Types of scripts](#)

[Scopes and scripts](#)

[Parameters add flexibility](#)

[Adding some logic](#)

[Conditional logic \(IF\)](#)

[Looping logic](#)

[More logic](#)

[Profiles](#)

[Summary](#)

[For further reading](#)

[7. Functions](#)

[Another kind of container](#)

[Comparing scripts and functions](#)

[Executing and calling functions](#)

[Naming conventions](#)

[Comment-based help](#)

[Parameters revisited](#)

[Typed parameters](#)

[Switches](#)

[Default values for parameters](#)

[Output](#)

[Summary](#)

[For further reading](#)

[8. Modules](#)

[Packaging functions](#)

[Script modules](#)

[The Export-ModuleMember cmdlet](#)

[Where do modules live?](#)

[Removing a module](#)

[PowerShell module autoloading](#)

[The #Requires statement](#)

[Removing a module – take two](#)

[Manifest modules](#)

[Listing modules](#)

[Summary](#)

[9. File I/O](#)

[Reading and writing text files](#)

[Writing text files](#)

[Working with CSV files](#)

[Output to CSV for quick reports](#)

[The Invoke-Item cmdlet](#)

[Import from CSV for quick objects](#)

[PowerShell streams and redirection](#)

[Other types of redirection operators](#)

[The out-file cmdlet](#)

[CLIXML – a special type of XML](#)

[Summary](#)

[For further reading](#)

[10. WMI and CIM](#)

[What is WMI?](#)

[WMI organization](#)

[Finding WMI classes](#)

[Retrieving objects with Get-WMIOObject](#)

[Getting the right instance](#)

[WQL syntax](#)

[Calling methods](#)

[WMI and CIM](#)

[The CIM cmdlets](#)

[CDXML modules](#)

[Summary](#)

[For further reading](#)

[11. Web Server Administration](#)

Installing IIS

[Detecting and installing IIS in Windows 8.1](#)

[Detecting and installing IIS in Server 2012R2](#)

Verifying IIS

[The WebAdministration module](#)

[Starting, stopping, and restarting IIS](#)

[Creating virtual directories and web applications](#)

[Working with application pools](#)

[Creating application pools](#)

[Summary](#)

[For further reading](#)

A. Next Steps

2. Module 2

1. Getting Started with Windows PowerShell

[Scripting the cmdlet style](#)

[Introducing Windows PowerShell](#)

[Installing Windows Management Framework 5.0](#)

[The Windows PowerShell consoles](#)

[The Windows PowerShell console host](#)

[Setting up the console host using GUI](#)

[Setting up the console host using PowerShell](#)

[Exploring the Windows PowerShell ISE host using GUI](#)

[Benefits of an ISE](#)

[The PowerShell ISE script browser](#)

[Using an interactive shell](#)

[Windows PowerShell cmdlets](#)

[Getting help](#)

[Understanding aliases](#)

[Understanding expressions](#)

[Understanding objects](#)

[Understanding pipelines](#)

[Exporting a running process to a CSV file](#)

[Understanding filtering and formatting](#)

[PowerShell formatting](#)

[Exploring snippets in the PowerShell ISE](#)

[Getting started with PowerShell scripting](#)

Summary

2. Unleashing Development Skills Using Windows PowerShell 5.0

Basics of WMI and CIM

Working with XML and COM

Exploring COM and Automation

Exploring .NET objects

Creating .NET objects

Extending .NET objects for Administrations and Development

tasks

Extending the .NET Framework types

Building advanced scripts and modules

Exploring Windows PowerShell 5.0

The basics of Desired State Configuration

The Authoring phase

The Staging phase

The "Make it so" phase

Use case of classes in WMF 5.0

Constructors

Parsing structured objects using PowerShell

Exploring Package Management

Exploring PowerShellGet

Understanding PowerShell modules

Introduction to modules

Script modules

Binary modules

Manifest modules

Dynamic modules

Script debugging

Managing breakpoints

Line breakpoints

Variable breakpoints

Command breakpoints

Debugging scripts

Summary

3. Exploring Desired State Configuration

Prerequisites

[Installing the WMF 5.0 April 2015 preview](#)
[Imperative versus declarative programming](#)
[Getting started with DSC](#)

[The Authoring phase](#)
[The Staging phase](#)
[The "Make it so" phase](#)
[Local Configuration Manager](#)
[Parameterizing the configuration script](#)
[Understanding MOF](#)

[Exploring Windows Remote Management and CIM](#)
[Windows PowerShell remoting](#)
[Exploring WSMAN cmdlets](#)
[HTTP/HTTPS Listener](#)
[Exploring CIM commands](#)
[Exploring CIM methods](#)
[Querying the remote machines using CIM](#)

[Creating configuration scripts](#)
[Creating a configuration with MOF](#)
[Creating a Class-defined DSC resource](#)

[Types of deployment modes](#)
[The push mode](#)
[The pull mode](#)
[Creating a pull server using the SMB share](#)
[Creating the pull server using HTTP and HTTPS](#)

[Summary](#)

[4. PowerShell and Web Technologies](#)

[PowerShell Web Access](#)
[Installing PowerShell Web Access](#)
[Configuring PowerShell Web Access](#)
[Applying authorization rules](#)

[Management OData IIS Extensions](#)
[Creating the Management OData web service](#)

[Exploring web requests](#)
[Downloading files from the Internet](#)
[Reading a file from the Internet](#)

[Exploring web services](#)

[Using web services](#)

[Building web services](#)

[Exploring the REST API](#)

[Using the Azure REST API in PowerShell](#)

[Exploring JSON](#)

[Summary](#)

[5. Exploring Application Programming Interface](#)

[Exploring API using PowerShell](#)

[The EWS API for managing Exchange Online](#)

[Purging items in the mailbox folder](#)

[Deleting items from the mailbox folder](#)

[The Lync 2013 client-side API](#)

[Installation of LYNC SDK](#)

[Exploring client settings](#)

[Automating test calls](#)

[IM with contacts](#)

[Client-side object model – SharePoint Online](#)

[How does CSOM Work?](#)

[Creating and deleting list](#)

[Making PowerShell modules with SDKs](#)

[Summary](#)

[3. Module 3](#)

[1. Variables, Arrays, and Hashes](#)

[Variables](#)

[Objects stored in variables](#)

[Arrays](#)

[Single-dimension arrays](#)

[Jagged arrays](#)

[Updating array values](#)

[Hashes](#)

[Deciding the best container for your scripts](#)

[Summary](#)

[2. Data Parsing and Manipulation](#)

[String manipulation](#)

[Replacing and splitting strings](#)

[Counting and trimming strings](#)

[The Trim method](#)
[The Substring method](#)
[The string true and false methods](#)
[Number manipulation and parsing](#)
[Formatting numbers](#)
[Formatting bytes](#)
[Date and time manipulation](#)
[Forcing data types](#)
[Piping variables](#)
[Summary](#)

[3. Comparison Operators](#)

[Comparison operator basics](#)
[Equal and not equal comparison](#)
[Greater than and less than comparison](#)
[Contains, like, and match operators](#)
[And / OR comparison operators](#)
[Best practices for comparison operators](#)
[Summary](#)

[4. Functions, Switches, and Loops Structures](#)

[Functions](#)
[Looping structures](#)
[Switches](#)
[Combining the use of functions, switches, and loops](#)
[Best practices for functions, switches, and loops](#)
[Best practices for functions](#)
[Best practices for looping structures and switches](#)
[Summary](#)

[5. Regular Expressions](#)

[Getting started with regular expressions](#)
[Regular expression grouping constructs and ranges](#)
[Regular expression quantifiers](#)
[Regular expression anchors](#)
[Regular expressions examples](#)
[Summary](#)

[6. Error and Exception Handling and Testing Code](#)

[Error and exception handling – parameters](#)

Error and exception handling – Try/Catch

Error and exception handling – Try/Catch with parameters

Error and exception handling – legacy exception handling

Methodologies for testing code

Testing the –WhatIf argument

Testing the frequency

Hit testing containers

Don't test in production

Summary

7. Session-based Remote Management

Utilizing CIM sessions

Creating a session

Creating a session with session options

Using sessions for remote management

Removing sessions

Summary

8. Managing Files, Folders, and Registry Items

Registry provider

Creating files, folders, and registry items with PowerShell

Adding named values to registry keys

Verifying files, folders, and registry items

Copying and moving files and folders

Renaming files, folders, registry keys, and named values

Deleting files, folders, registry keys, and named values

Summary

9. File, Folder, and Registry Attributes, ACLs, and Properties

Retrieving attributes and properties

Viewing file and folder extended attributes

Setting the mode and extended file and folder attributes

Managing file, folder, and registry permissions

Copying access control lists

Adding and removing ACL rules

Summary

10. Windows Management Instrumentation

WMI structure

Using WMI objects

[Searching for WMI classes](#)

[Creating, modifying, and removing WMI property instances](#)

[Creating property instances](#)

[Modifying property instances](#)

[Removing property instances](#)

[Invoking WMI class methods](#)

[Summary](#)

[11. XML Manipulation](#)

[XML file structure](#)

[Reading XML files](#)

[Adding XML content](#)

[Modifying XML content](#)

[Removing XML content](#)

[Summary](#)

[12. Managing Microsoft Systems with PowerShell](#)

[Managing local users and groups](#)

[Managing local users](#)

[Managing local groups](#)

[Querying for local users and groups](#)

[Managing Windows services](#)

[Managing Windows processes](#)

[Installing Windows features and roles](#)

[Summary](#)

[13. Automation of the Environment](#)

[Invoking programs for automation](#)

[Using desired state configuration](#)

[Authoring phase](#)

[Staging and remediation phase](#)

[Detecting and restoring drifting configurations](#)

[Summary](#)

[14. Script Creation Best Practices and Conclusion](#)

[Best practices for script management](#)

[# commenting headers](#)

[Commenting code](#)

[Best practices for script creation](#)

[Script structure](#)

[Other important best practices for script creation](#)

[Controlling source files](#)

[Best practices for software automation](#)

[Summary](#)

[Mastering Windows PowerShell Scripting – conclusion](#)

[Staying connected with the author](#)

[Bibliography](#)

[Index](#)

PowerShell: Automating Administrative Tasks

PowerShell: Automating Administrative Tasks

Learn PowerShell from the inside out, right from basic scripting all the way to becoming a master at automating, managing, and maintaining your Windows environment

A course in three modules



BIRMINGHAM - MUMBAI

PowerShell: Automating Administrative Tasks

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: February 2017

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-375-5

www.packtpub.com

Credits

Author

Michael Shepard

Chendrayan Venkatesan

Sherif Talaat

Brenton J.W. Blawat

Reviewers

Richard Gibson

David Green

Keith Lindsay

Mayur Arvind Makwana

Ashley Poole

Laxmikant P Patil

Tim Amico

Christophe CRÉMON

Tomas Restrepo

Content Development Editor

Amedh Pohad

Graphics

Kirk D'Penha

Production Coordinator

Shantanu Zagade

Preface

Windows PowerShell is a task-based command-line shell and scripting language designed specifically for system administration. Built on the .NET Framework, Windows PowerShell helps IT professionals and power users control and automate the administration of the Windows operating system and applications that run on Windows.

PowerShell is quickly becoming the de facto standard for scripting in Microsoft Windows environments. It enables the automation of otherwise complex tasks, providing interactivity between different products.

What this learning path covers

[Module 1](#), *Getting Started with PowerShell*, helps you get up and running with PowerShell, taking you from the basics of installation, to writing scripts and web server automation. This module, as an introduction to the central topics of PowerShell, covers finding and understanding PowerShell commands and packaging code for reusability, right through to a practical example of automating IIS. It also includes topics such as installation and setup, creating scripts, automating tasks, and using Powershell to access data stores, registry, and file systems. You will explore the PowerShell environment and discover how to use cmdlets, functions, and scripts to automate Windows systems. Along the way, you will learn to perform data manipulation and solve common problems using basic file input/output functions. By the end of this module, you will be familiar with PowerShell and be able to utilize the lessons learned from the module to automate your servers.

[Module 2](#), *Windows PowerShell for .NET Developers - Second Edition*, begins with the Windows PowerShell basics, explores the significant features of Windows Management Framework 5.0, covers the basic concepts of Desired State Configuration and the importance of

idempotent deployments. By the end of the module, you will have a good understanding of Windows PowerShell's features and will be able to automate your tasks and manage configuration effectively.

[Module 3](#), *Mastering Windows PowerShell Scripting*, beginning with PowerShell fundamentals, this module progresses by imparting the advanced skills required to master automation. You will learn how to deal with data and operations on various data types and structures, and see the techniques of data manipulation and parsing. Moving on, you will learn about the usage of regular expressions and comparison operators. Error handling techniques will enable you to identify and eliminate errors. This module also provides best practices for scripting and techniques to reduce the amount of code required to complete tasks. By the end of this module, you will be able to create a variety of PowerShell scripts and successfully automate your environment and become the go-to person.

What you need for this learning path

For Module 1:

Most of the examples in the module will work with PowerShell Version 2.0 and above. In the sections where a higher version of the engine is required, it will be indicated in the text. You should have no problems running the provided code on either a client installation (Windows 7 or greater) or a server installation (Windows Server 2008 R2 or higher).

For Module 2:

The following is a list of software required and supported operating systems for this module:

- Windows Management Framework 3.0 and advanced (Windows 7)
- Windows Management Framework 5.0 (Windows 7, Windows Server 2008 R2 SP1 and advanced)
- Windows Management Framework 5.0 (Windows 7, Windows Server 2012 R2)

For Module 3:

To work through the examples provided in this module, you will need access to Windows 7 or a higher Windows operating system. You will also need Server 2008 R2 or a higher Windows Server operating system. The chapters in this module rely highly on Windows Management Framework 4.0 (PowerShell 4.0) and Remote Server Administration Tools. You will need to download and install both of these software packages on the systems you are running these examples on.

Who this learning path is for

If you are a system administrator who wants to become an expert in automating and managing your Windows environment, then this course is for you. Some basic understanding of PowerShell would be helpful.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a course, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/PowerShell-Automating-Administrative-Tasks>. We also have other code bundles from our rich catalog of books, courses and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to

bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Part 1. Module 1

Getting Started with PowerShell

Learn the fundamentals of PowerShell to build reusable scripts and functions to automate administrative tasks with Windows

Chapter 1. First Steps

In the world of Microsoft system administration, it is becoming impossible to avoid PowerShell. If you've never looked at PowerShell before or want a refresher to make sure you understand what is going on, this chapter would be a good place to start. We will cover the following topics:

- Determining the installed PowerShell version
- Installing/Upgrading PowerShell
- Starting a PowerShell session
- Simple PowerShell commands
- PowerShell Aliases

Determining the installed PowerShell version

There are many ways to determine which version of PowerShell, if any, is installed on a computer. We will examine how this information is stored in the registry and how to have PowerShell tell you the version.

Tip

It's helpful while learning new things to actually follow along on the computer as much as possible. A big part of the learning process is developing "muscle memory", where you get used to typing the commands you see throughout Module 1. As you do this, it will also help you pay closer attention to the details in the code you see, since the code won't work correctly if you don't get the syntax right.

Using the registry to find the installed version

If you're running at least Windows 7 or Server 2008, you already have PowerShell installed on your machine. Windows XP and Server 2003 can install PowerShell, but it isn't pre-installed as part of the **Operating System (OS)**. To see if PowerShell is installed at all, no matter what OS

is running, inspect the following registry entry:

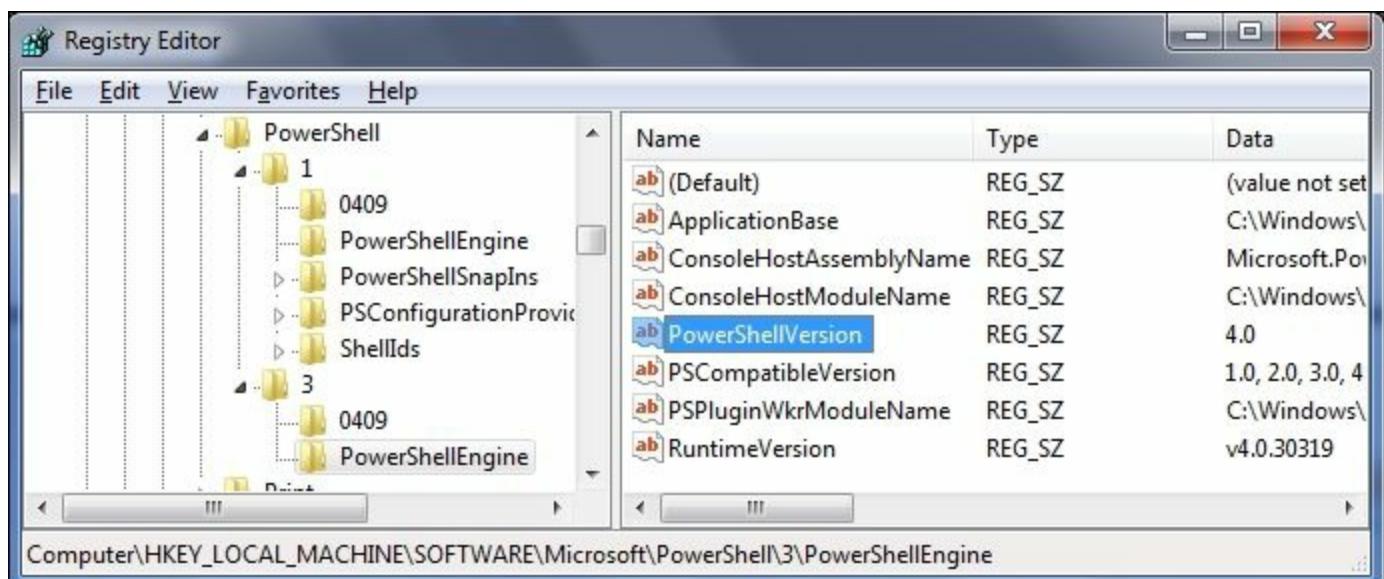
HKLM\Software\Microsoft\PowerShell\1\Install

If this exists and contains the value of 1, PowerShell is installed. To see, using the registry, what version of PowerShell is installed, there are a couple of places to look at.

First, the following value will exist if PowerShell 3.0 or higher is installed:

HKLM\SOFTWARE\Microsoft\PowerShell\3\PowerShellEngine\PowerShellVersion

If this value exists, it will contain the PowerShell version that is installed. For instance, my Windows 7 computer is running PowerShell 4.0, so `regedit.exe` shows the value of 4.0 in that entry:



If this registry entry does not exist, you have either PowerShell 1.0 or 2.0 installed. To determine which, examine the following registry entry (note the 3 is changed to 1):

HKLM\SOFTWARE\Microsoft\PowerShell\1\PowerShellEngine\PowerShellVersion

This entry will either contain 1.0 or 2.0, to match the installed version of

PowerShell.

It is important to note that all versions of PowerShell after 2.0 include the 2.0 engine, so this registry entry will exist for any version of PowerShell. For example, even if you have PowerShell 5.0 installed, you will also have the PowerShell 2.0 engine installed and will see both 5.0 and 2.0 in the registry.

Using PowerShell to find the installed version

No matter what version of PowerShell you have installed, it will be installed in the same place. This installation directory is called `PSHOME` and is located at:

```
%WINDIR%\System32\WindowsPowerShell\v1.0\
```

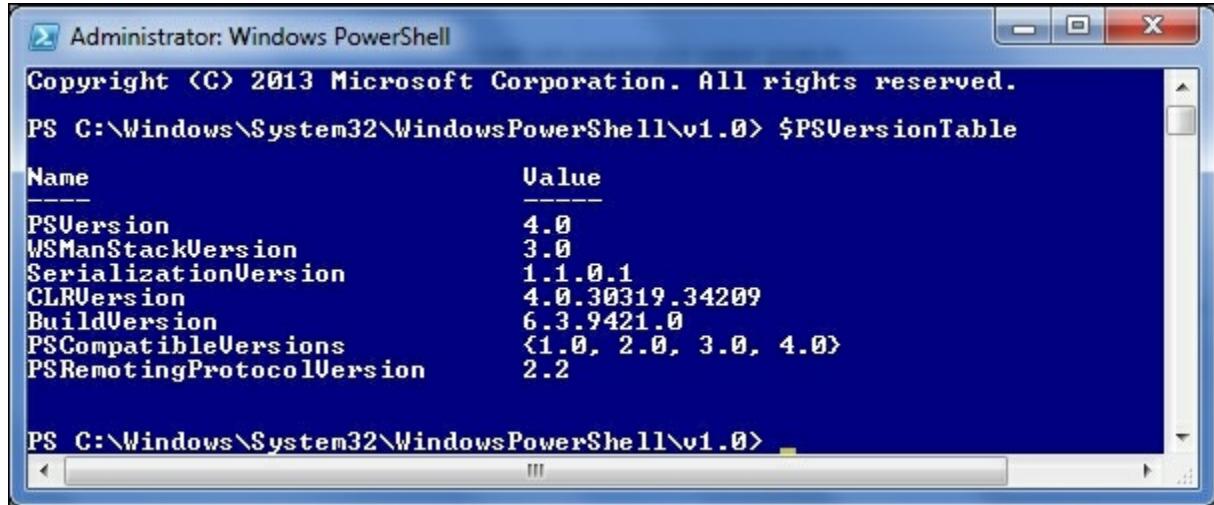
In this folder, there will be an executable file called `PowerShell.exe`. There should be a shortcut to this in your start menu, or in the start screen, depending on your operating system. In either of these, search for PowerShell and you should find it. Running this program opens the PowerShell console, which is present in all the versions.

At first glance, this looks like Command Prompt but the text (and perhaps the color) should give a clue that something is different:



In this console, type `$PSVersionTable` in the command-line and press

Enter. If the output is an error message, PowerShell 1.0 is installed, because the \$PSVersionTable variable was introduced in PowerShell 2.0. If a table of version information is seen, the PSVersion entry in the table is the installed version. The following is the output on a typical computer, showing that PowerShell 4.0 is installed:



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "PS C:\Windows\System32\WindowsPowerShell\v1.0> \$PSVersionTable" is run, and the output is displayed as a table:

Name	Value
PSVersion	4.0
WSManStackVersion	3.0
SerializationVersion	1.1.0.1
CLRVersion	4.0.30319.34209
BuildVersion	6.3.9421.0
PSCOMPATIBLEVERSIONS	{1.0, 2.0, 3.0, 4.0}
PSRemotingProtocolVersion	2.2

PS C:\Windows\System32\WindowsPowerShell\v1.0>

Tip

Warning

You might find some references on the Internet telling you to use the `get-host` cmdlet to easily find the PowerShell version. Unfortunately, this only tells you the version of the PowerShell host, that is, the program you're using to run PowerShell.

Installing/upgrading PowerShell

If you don't have PowerShell installed or want a more recent version of PowerShell, you'll need to find the **Windows Management Framework (WMF)** download that matches the PowerShell version you want. WMF includes PowerShell as well as other related tools such as **Windows Remoting (WinRM)**, **Windows Management Instrumentation (WMI)**, and **Desired State Configuration (DSC)**. The contents of the distribution change from version to version, so make sure to read the release notes included in the download. Here are links to the installers:

PowerShell Version	URL
1.0	http://support.microsoft.com/kb/926139
2.0	http://support2.microsoft.com/kb/968929/en-us
3.0	http://www.microsoft.com/en-us/download/details.aspx?id=34595
4.0	http://www.microsoft.com/en-us/download/details.aspx?id=40855
5.0 (Feb. Preview)	http://www.microsoft.com/en-us/download/details.aspx?id=45883

Note that PowerShell 5.0 has not been officially released, so the table lists the February 2015 preview, the latest at the time of writing.

The PowerShell 1.0 installer was released as an executable (.exe), but since then the releases have all been as standalone Windows update installers (.msu). All of these are painless to execute. You can simply download the file and run it from the explorer or from the **Run...** option in the start menu. PowerShell installs don't typically require a reboot but it's best to plan on doing one, just in case.

It's important to note that you can only have one version of PowerShell installed, and you can't install a lower version than the version that was shipped with your OS. Also, there are noted compatibility issues

between various versions of PowerShell and Microsoft products such as Exchange, System Center, and Small Business Server, so make sure to read the system requirements section on the download page. Most of the conflicts can be resolved with a service pack of the software, but you should be sure of this before upgrading PowerShell on a server.

Starting a PowerShell session

Technet24.ir

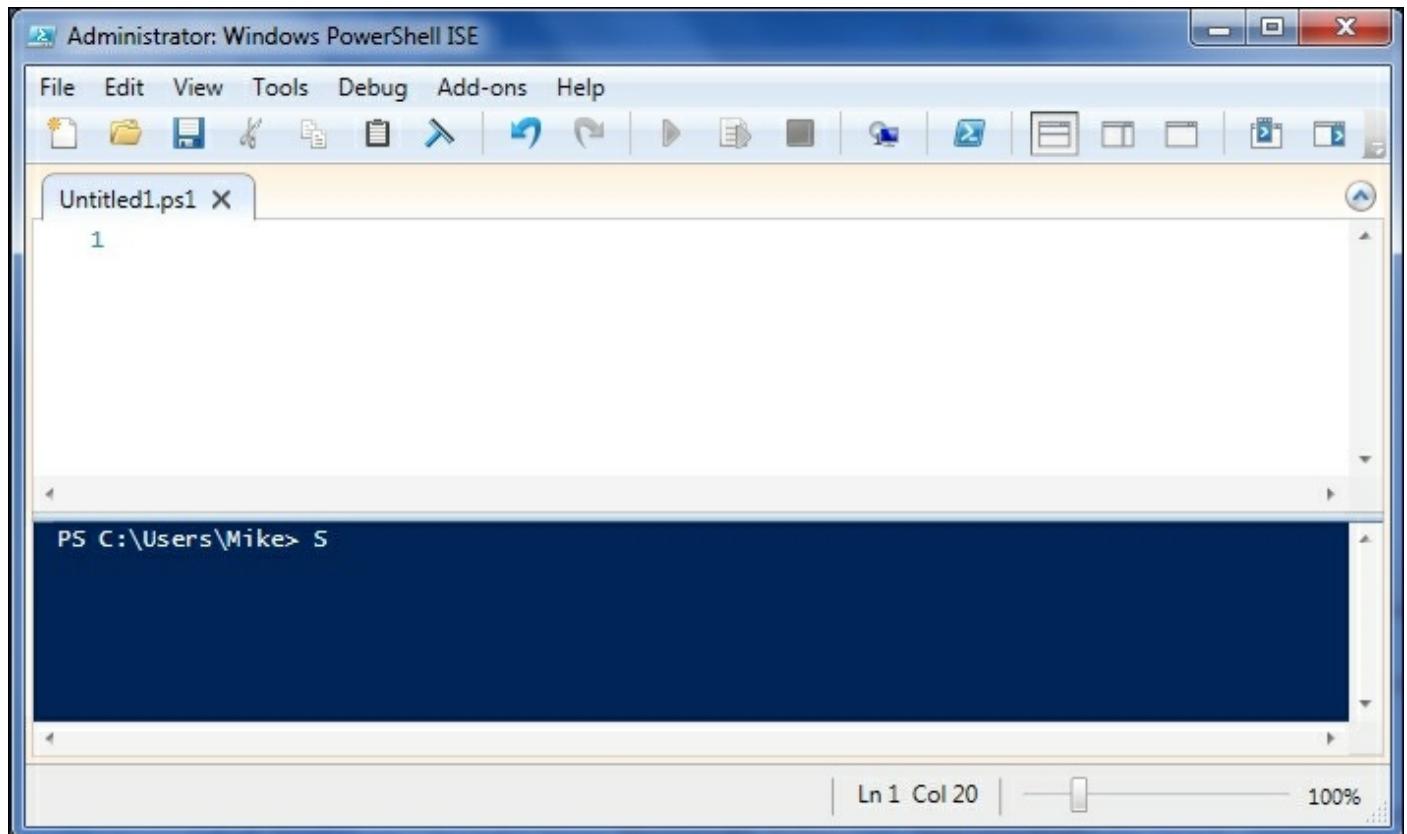
We already started a PowerShell session earlier in the section on using PowerShell to find the installed version. So, what more is there to see? It turns out that there is more than one program used to run PowerShell, possibly more than one version of each of these programs, and finally, more than one way to start each of them. It might sound confusing but it will all make sense shortly.

PowerShell hosts

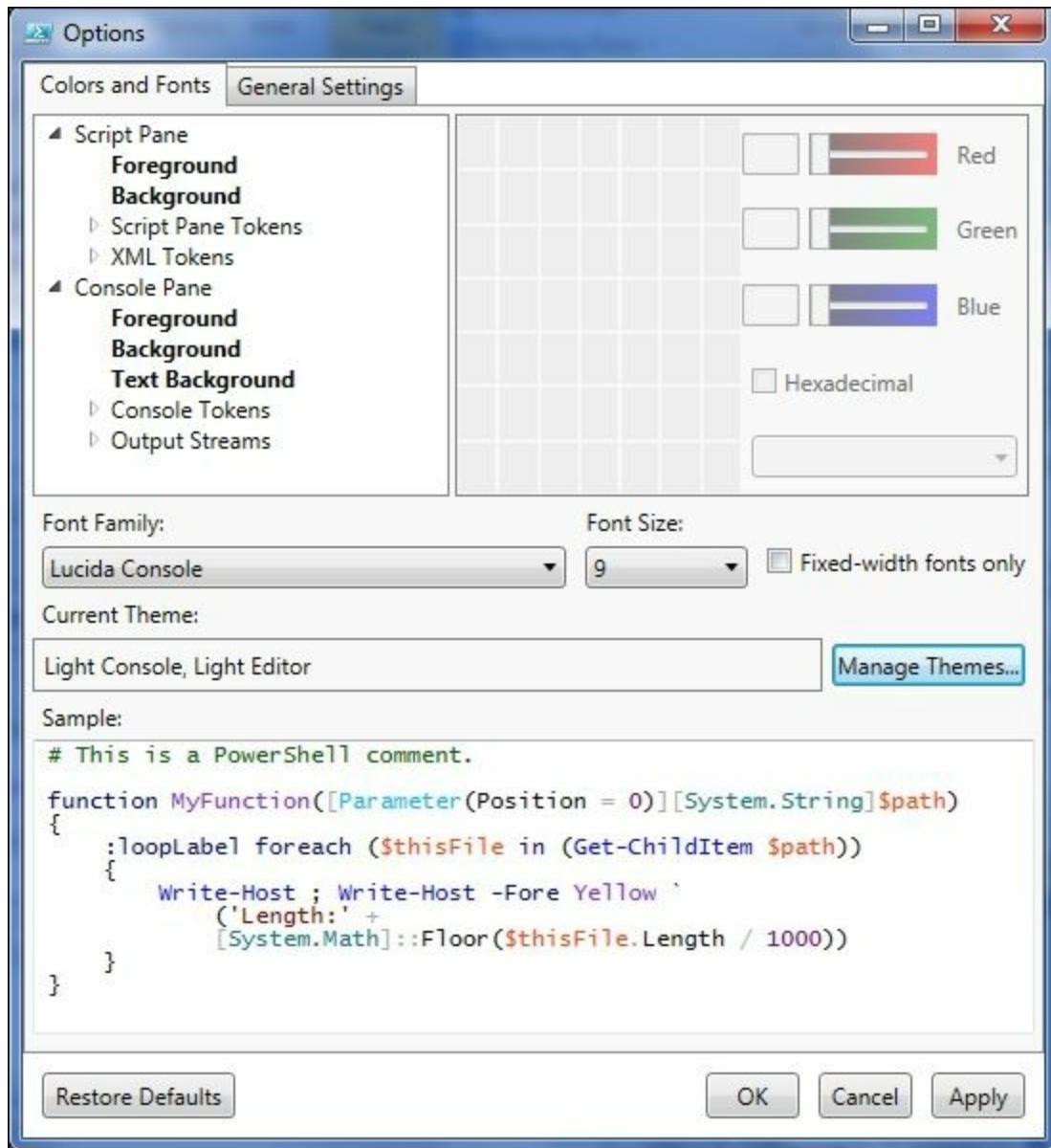
A PowerShell host is a program that provides access to the PowerShell engine in order to run PowerShell commands and scripts. The `PowerShell.exe` that we saw in the `PSHOME` directory earlier in this chapter is known as the **console host**. It is cosmetically similar to Command Prompt (`cmd.exe`) and only provides a command-line interface. Starting with Version 2.0 of PowerShell, a second host was provided.

The **Integrated Scripting Environment (ISE)** is a graphical environment providing multiple editors in a tabbed interface along with menus and the ability to use plugins. While not as fully featured as an **Integrated Development Environment (IDE)**, the ISE is a tremendous productivity tool used to build PowerShell scripts and is a great improvement over using an editor, such as notepad for development.

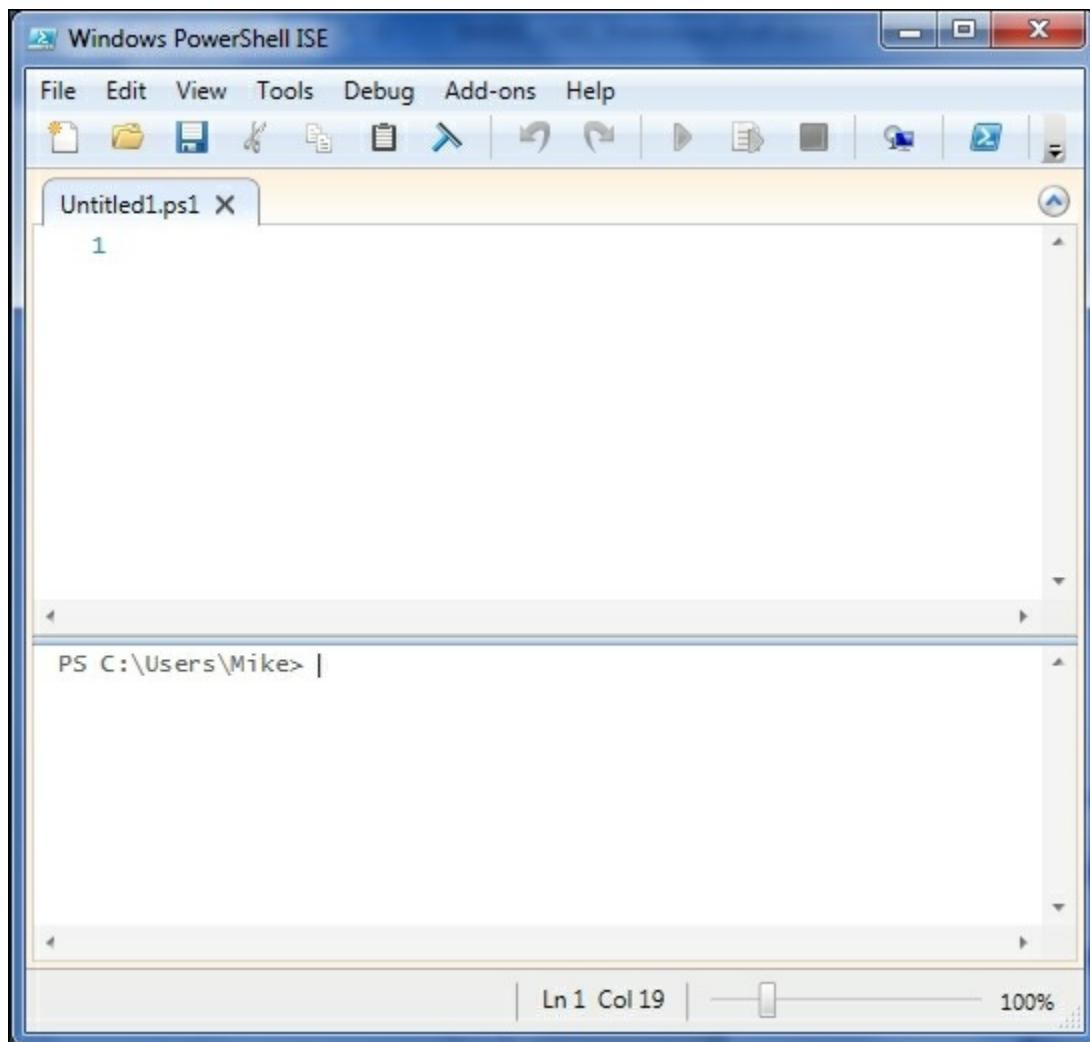
The ISE executable is stored in `PSHOME`, and is named `powershell_ise.exe`. In Version 2.0 of the ISE, there were three sections, a tabbed editor, a console for input, and a section for output. Starting with Version 3.0, the input and output sections were combined into a single console that is more similar to the interface of the console host. The Version 4.0 ISE is shown as follows:



I will be using the **Light Console, Light Editor** theme for the ISE in most of the screenshots for Module 1, because the dark console does not work well on the printed page. To switch to this theme, open the **Options** item in the **Tools** Menu and select **Manage Themes...** in the options window:



Press the **Manage Themes...** button, select the **Light Console, Light Editor** option from the list and press **OK**. Press **OK** again to exit the options screen and your ISE should look something similar to the following:



Note that you can customize the appearance of the text in the editor and the console pane in other ways as well. Other than switching to the light console display, I will try to keep the settings to default.

64-bit and 32-bit PowerShell

In addition to the console host and the ISE, if you have a 64-bit operating system, you will also have 64-bit and 32-bit PowerShell installations that will include separate copies of both the hosts.

As mentioned before, the main installation directory, or `PSHOME`, is found at `%WINDIR%\System32\WindowsPowerShell\v1.0\`. The version of PowerShell in `PSHOME` matches that of the the operating system. In other words, on a 64-bit OS, the PowerShell in `PSHOME` is 64-bit. On a 32-bit system, `PSHOME` has a 32-bit PowerShell install. On a 64-bit system, a

second 32-bit system is found in

%WINDIR%\SysWOW64\WindowsPowerShell\v1.0.

Tip

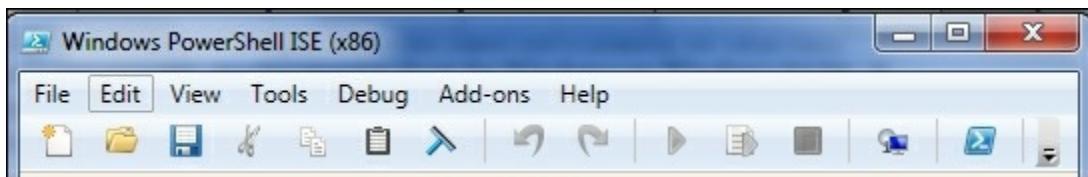
Isn't that backward?

It seems backward that the 64-bit install is in the System32 folder and the 32-bit install is in SysWOW64. The System32 folder is always the primary system directory on a Windows computer, and this name has remained for backward compatibility reasons. SysWOW64 is short for Windows on Windows 64-bit. It contains the 32-bit binaries required for 32-bit programs to run in a 64-bit system, since 32-bit programs can't use the 64-bit binaries in System32.

Looking in the Program Files\Accessories\Windows PowerShell menu in the start menu of a 64-bit Windows 7 install, we see the following:



Here, the 32-bit hosts are labeled as (x86) and the 64-bit versions are undesignated. When you run the 32-bit hosts on a 64-bit system, you will also see the (x86) designation in the title bar:



PowerShell as an administrator

When you run a PowerShell host, the session is not elevated. This means

that even though you might be an administrator of the machine, the PowerShell session is not running with administrator privileges. This is a safety feature to help prevent users from inadvertently running a script that damages the system.

In order to run a PowerShell session as an administrator, you have a couple of options. First, you can right-click on the shortcut for the host and select **Run as administrator** from the context menu. When you do this, unless you have disabled the UAC alerts, you will see a **User Account Control (UAC)** prompt verifying whether you want to allow the application to run as an administrator.



Selecting **Yes** allows the program to run as an administrator, and the title bar reflects that this is the case:



The second way to run one of the hosts as an administrator is to right-click on the shortcut and choose **Properties**. On the shortcut tab of the properties window, press the **Advanced** button. In the **Advanced Properties** window that pops up, check the **Run as administrator** checkbox and press **OK**, and **OK** again to exit out of the properties window:



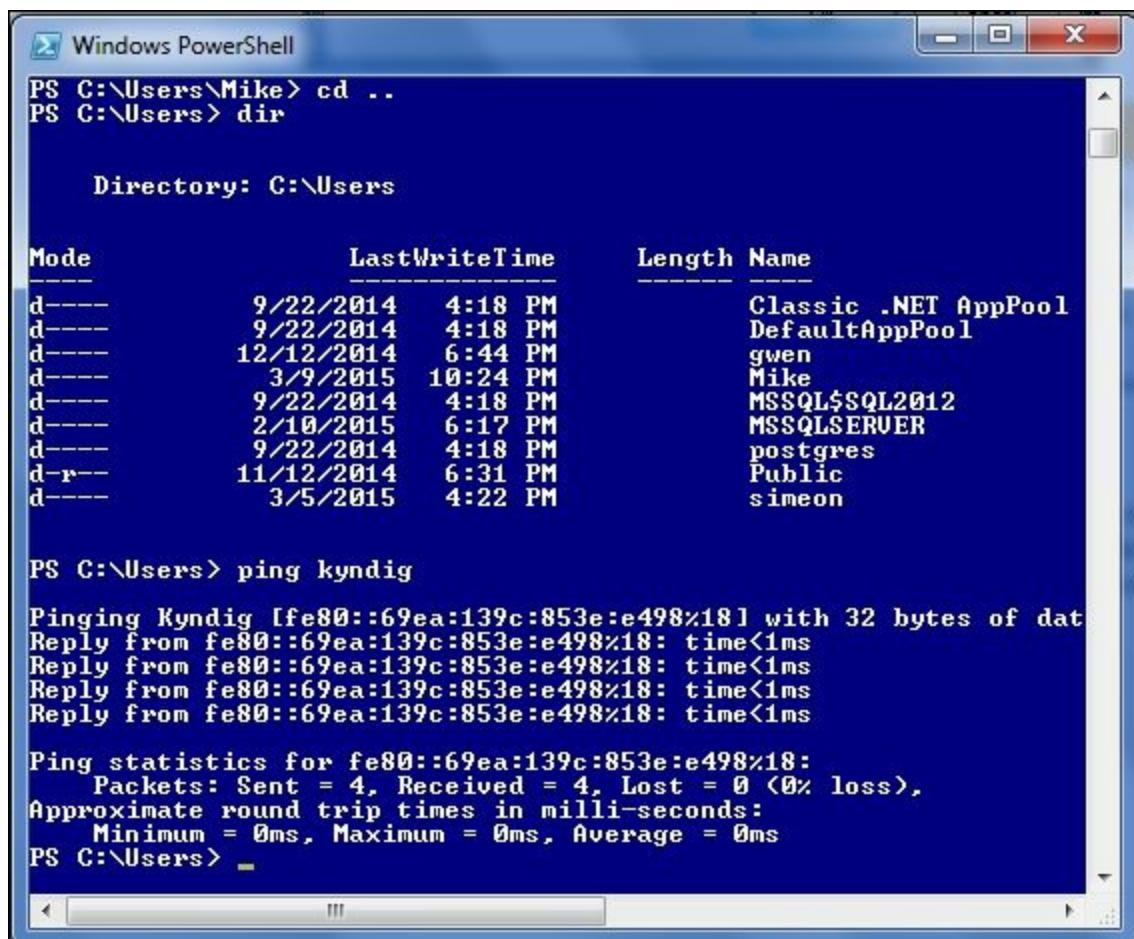
Using this technique will cause the shortcut to always launch as an administrator, although the UAC prompt will still appear.

Tip

If you choose to disable UAC, PowerShell hosts always run as administrators. Note that disabling UAC alerts is not recommended.

Simple PowerShell commands

Now that we know all the ways that can get a PowerShell session started, what can we do in a PowerShell session? I like to introduce people to PowerShell by pointing out that most of the command-line tools that they already know work fine in PowerShell. For instance, try using `DIR`, `CD`, `IPCONFIG`, and `PING`. Commands that are part of Command Prompt (think DOS commands) might work slightly different in PowerShell if you look closely, but typical command-line applications work exactly the same as they have always worked in Command Prompt:



The screenshot shows a Windows PowerShell window titled "Windows PowerShell". The session starts with the command `cd ..` followed by `dir`, which lists the contents of the `C:\Users` directory. The output includes columns for Mode, LastWriteTime, Length, and Name, showing various user profiles and system folders. Below this, the command `ping kyndig` is run, followed by its output showing four successful replies from the target IP address. Finally, the command `get-verb` is run to list available verbs.

```
PS C:\Users\Mike> cd ..
PS C:\Users> dir

    Directory: C:\Users

Mode                LastWriteTime         Length Name
----                -----         -----   -----
d----        9/22/2014     4:18 PM          -----   Classic .NET AppPool
d----        9/22/2014     4:18 PM          -----   DefaultAppPool
d----       12/12/2014    6:44 PM          -----   gwen
d----        3/9/2015     10:24 PM         -----   Mike
d----        9/22/2014     4:18 PM          -----   MSSQL$SQL2012
d----       2/10/2015     6:17 PM          -----   MSSQLSERVER
d----        9/22/2014     4:18 PM          -----   postgres
d-r--      11/12/2014    6:31 PM          -----   Public
d----        3/5/2015     4:22 PM          -----   simeon

PS C:\Users> ping kyndig
Pinging Kyndig [fe80::69ea:139c:853e:e498%18] with 32 bytes of data
Reply from fe80::69ea:139c:853e:e498%18: time<1ms
Reply from fe80::69ea:139c:853e:e498%18: time<1ms
Reply from fe80::69ea:139c:853e:e498%18: time<1ms
Reply from fe80::69ea:139c:853e:e498%18: time<1ms

Ping statistics for fe80::69ea:139c:853e:e498%18:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
PS C:\Users> get-verb
```

PowerShell commands, called `cmdlets`, are named with a verb-noun convention. Approved verbs come from a list maintained by Microsoft and can be displayed using the `get-verb` cmdlet:

Verb	Group
Add	Common
Clear	Common
Close	Common
Copy	Common
Enter	Common
Exit	Common

By controlling the list of verbs, Microsoft has made it easier to learn PowerShell. The list is not very long and it doesn't contain verbs that have the same meaning (such as Stop, End, Terminate, and Quit), so once you learn a cmdlet using a specific verb, you can easily guess the meaning of the cmdlet names that include the verb.

Some other easy to understand cmdlets are:

- Clear-Host (clears the screen)
- Get-Date (outputs the date)
- Start-Service (starts a service)
- Stop-Process (stops a process)
- Get-Help (shows help about something)

Note that these use several different verbs. From this list, you can probably guess what cmdlet you would use to stop a service. Since you know there's a `Start-Service` cmdlet, and you know from the `Stop-Process` cmdlet that `Stop` is a valid verb, it is logical that `Stop-Service` is what you would use. The consistency of PowerShell cmdlet naming is a tremendous benefit to learners of PowerShell, and it is a policy that is important as you write the PowerShell code.

Tip

What is a cmdlet?

The term cmdlet was coined by Jeffery Snover, the inventor of PowerShell to refer to the PowerShell commands. The PowerShell commands aren't particularly different from other commands, but by

giving a unique name to them, he ensured that PowerShell users would be able to use search engines to easily find PowerShell code simply by including the term cmdlet.

PowerShell aliases

If you tried to use `DIR` and `CD` in the last section, you may have noticed that they didn't work exactly as the DOS commands that they resemble. In case you didn't see this, enter `DIR /S` on a PowerShell prompt and see what happens. You will either get an error complaining about a path not existing, or get a listing of a directory called `s`. Either way, it's not the listing of files including subdirectories. Similarly, you might have noticed that `CD` in PowerShell allows you to switch between drives without using the `/D` option and even lets you change to a UNC path. The point is that, these are not DOS commands. What you're seeing is a PowerShell **alias**.

Aliases in PowerShell are alternate names that PowerShell uses for both PowerShell commands and programs. For instance, in PowerShell, `DIR` is actually an alias for the `Get-ChildItem` cmdlet. The `CD` alias points to the `Set-Location` cmdlet. Aliases exist for many of the cmdlets that perform operations similar to the commands in DOS, Linux, or Unix shells.

Aliases serve two main purposes in PowerShell, as follows:

- They allow more concise code on Command Prompt
- They ease users' transition from other shells to PowerShell

To see a list of all the aliases defined in PowerShell, you can use the `Get-Alias` cmdlet. To find what an alias references, type `Get-Alias <alias>`. For example, see the following screenshot:



```
PS C:\Users\Mike> get-alias DIR
 CommandType      Name
 -----          -----
 Alias           dir -> Get-ChildItem

```

To find out what aliases exist for a cmdlet, type `Get-Alias -Definition <cmdlet>` as follows:

```
PS C:\Users\Mike> get-alias -Definition Get-ChildItem
```

CommandType	Name	ModuleName
Alias	dir -> Get-ChildItem	
Alias	gci -> Get-ChildItem	
Alias	ls -> Get-ChildItem	

Here we can see that the `Get-ChildItem` cmdlet has three aliases. The first and last assist in transitioning from DOS and Linux shells, and the middle one is an abbreviation using the first letters in the name of the cmdlet.

Summary

This chapter focused on figuring out what version of PowerShell was installed and the many ways to start a PowerShell session. A quick introduction to PowerShell cmdlets showed that a lot of the command-line knowledge we have from DOS can be used in PowerShell and that aliases make this transition easier.

In the next chapter, we will look at the `Get-Command`, `Get-Help`, and `Get-Member` cmdlets, and learn how they unlock the entire PowerShell ecosystem for us.

For further reading

The Monad Manifesto, which outlines the original vision of the PowerShell project:

<http://blogs.msdn.com/b/powershell/archive/2007/03/19/monad-manifesto-the-origin-of-windows-powershell.aspx>

Microsoft's approved cmdlet verb list: <http://msdn.microsoft.com/en-us/library/ms714428.aspx>

- get-help about_aliases
- get-help about_PowerShell.exe
- get-help about_PowerShell_ISE.exe

Chapter 2. Building Blocks

Even though books, videos, and the Internet can be helpful in your efforts to learn PowerShell, you will find that your greatest ally in this quest is PowerShell itself. In this chapter, we will look at two fundamental weapons in any PowerShell scripter's arsenal, the `Get-Command` and `Get-Help` cmdlets. The topics covered in this chapter include the following:

- Finding commands using `Get-Command`
- Finding commands using tab completion
- Using `Get-Help` to understand cmdlets
- Interpreting the command syntax

What can you do?

You saw in the previous chapter that you are able to run standard command-line programs in PowerShell and that there are aliases defined for some cmdlets that allow you to use the names of commands that you are used to from other shells. Other than these, what can you use? How do you know which commands, cmdlets, and aliases are available?

The answer is the first of the **big three** cmdlets, the `Get-Command` cmdlet. Simply executing `Get-Command` with no parameters displays a list of all the entities that PowerShell considers to be executable. This includes programs in the path (the environment variable), cmdlets, functions, scripts, and aliases.

The screenshot shows the Windows PowerShell ISE interface. The title bar reads "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The main window displays the command "Get-Command" and its output. The output is a table with three columns: CommandType, Name, and ModuleName. The table lists numerous commands, mostly aliases and functions, from various modules like WebAdministration, PSDiagnostics, and PSDesiredStateConfiguration.

CommandType	Name	ModuleName
Alias	Begin-WebCommitDelay	WebAdministration
Alias	End-WebCommitDelay	WebAdministration
Function	A:	
Function	B:	
Function	C:	
Function	cd..	
Function	cd\	
Function	Clear-Host	PSDesiredStateConfiguration
Function	Configuration	
Function	D:	
Function	Disable-PSTrace	PSDiagnostics
Function	Disable-PSWSManCombinedTrace	PSDiagnostics
Function	Disable-WSManTrace	PSDiagnostics
Function	E:	
Function	Enable-PSTrace	PSDiagnostics
Function	Enable-PSWSManCombinedTrace	PSDiagnostics
Function	Enable-WSManTrace	PSDiagnostics

This list of commands is long and gets longer with each new operating system and PowerShell release. To count the output, we can use this command:

Get-Command | Measure-Object

The output of this command is as follows:

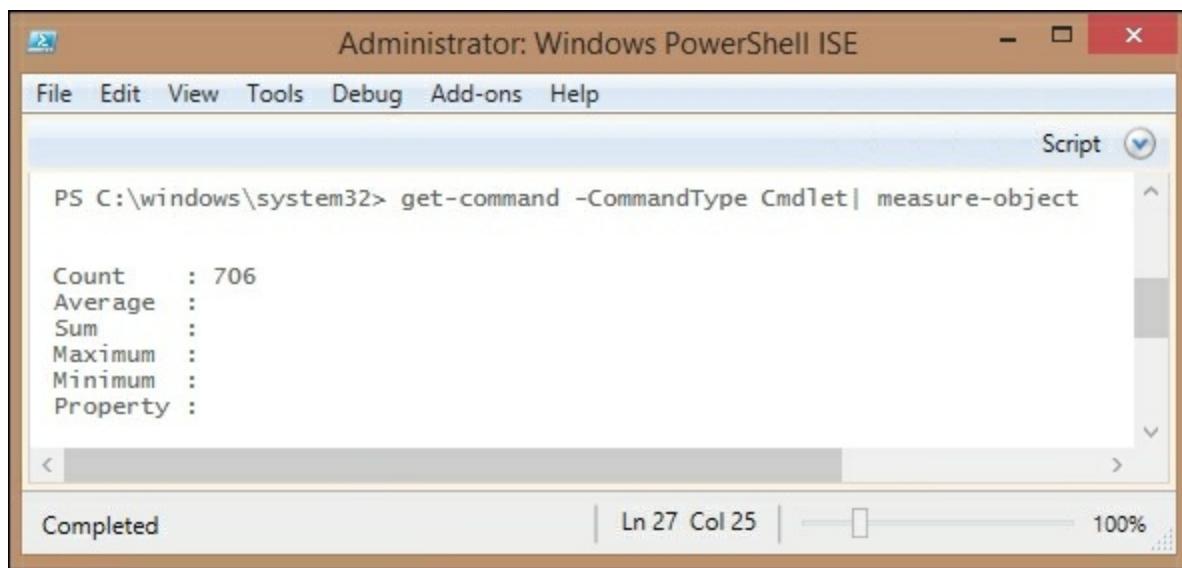
The screenshot shows the "Administrator: Windows PowerShell ISE" window. The title bar includes the administrator context ("Administrator: Windows PowerShell ISE"). The menu bar and toolbar are similar to the previous screenshot. The main window shows the command "get-command | measure-object" and its output. The output is a table with five rows: Count, Average, Sum, Maximum, and Minimum, all currently showing a value of ":".

Count	:
Average	:
Sum	:
Maximum	:
Minimum	:

Tip

The pipe character (|) tells PowerShell to use the output of the command on the left of the pipe as the input of the command on the right. In this example, it says that the output of the `get-command` cmdlet should be used as the input for the `measure-object` cmdlet.

The `Measure-Object` cmdlet can be used to count the number of items in the output from a command. In this case, it tells us that there are 1444 different commands available to us. In the output, we can see that there is a `CommandType` column. There is a corresponding parameter to the `Get-Command` cmdlet that allows us to filter the output and only shows us certain kinds of commands. Limiting the output to the cmdlets shows that there are 706 different cmdlets available:



A screenshot of the Windows PowerShell ISE window titled "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar has a dropdown labeled "Script" with a checkmark. The main pane displays the following PowerShell command and its output:

```
PS C:\windows\system32> get-command -CommandType Cmdlet | measure-object
```

Count	: 706
Average	:
Sum	:
Maximum	:
Minimum	:
Property	:

The status bar at the bottom shows "Completed" and "Ln 27 Col 25".

Getting a big list such as this isn't very useful though. In reality, you are usually going to use `Get-Command` to find commands that pertain to a component you want to work with. For instance, if you need to work with Windows services, you could issue a command such as `Get-Command *Service*` using wildcards to include any command that includes the word `service` in its name. The output would look something similar to the following:

The screenshot shows a Windows PowerShell ISE window with the title bar "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with a "Script" button is visible. The main pane displays the command "get-command *Service*" and its output:

```
PS C:\Users\Mike> get-command *Service*

CommandType      Name                           ModuleName
----           ----                         -----
Cmdlet          Get-Service                   Microsoft...
Cmdlet          New-Service                  Microsoft...
Cmdlet          New-WebServiceProxy          Microsoft...
Cmdlet          Restart-Service             Microsoft...
Cmdlet          Resume-Service              Microsoft...
Cmdlet          Set-Service                 Microsoft...
Cmdlet          Start-Service               Microsoft...
Cmdlet          Stop-Service                Microsoft...
Cmdlet          Suspend-Service            Microsoft...
Application     ControlService.exe          Microsoft...
Application     services.exe                Microsoft...
Application     services.msc              Microsoft...
```

The status bar at the bottom indicates "Ln 4079 Col 19" and "100%".

You can see in the output that applications found in the current path and the PowerShell cmdlets are both included. Unfortunately, there's also a cmdlet called `New-WebServiceProxy`, which clearly doesn't have anything to do with Windows services. To better control the matching of cmdlets with your subject, you can use the `-Noun` parameter to filter the commands. Note that by naming a noun, you are implicitly limiting the results to PowerShell objects because nouns are a PowerShell concept. Using this approach gives us a smaller list:

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

```
PS C:\Users\Mike> Get-Command -Noun Service
```

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

```
PS C:\Users\Mike> |
```

Ln 36 Col 19 | 100%

The scripter's secret weapon – tab completion

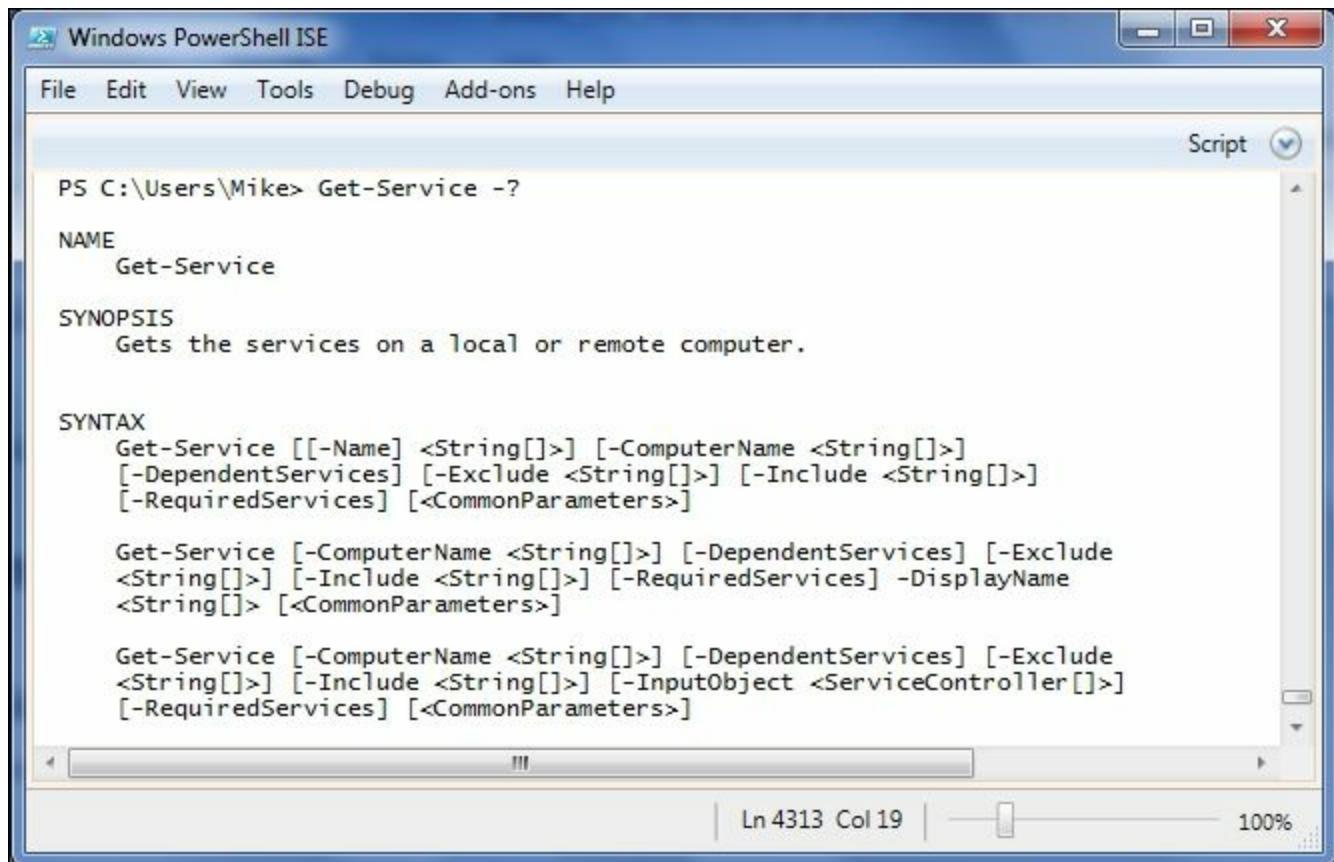
Although `Get-Command` is a great way to find cmdlets, the truth is that the PowerShell cmdlet names are very predictable. In fact, they're so predictable that after you've been using PowerShell for a while you won't probably turn to `Get-Command` very often. After you've found the noun or the set of nouns that you're working with, the powerful tab completion found in both the PowerShell console and the ISE will allow you to enter just a part of the command and press *tab* to cycle through the list of commands that match what you have entered. For instance, in keeping with our examples dealing with services, you could enter `*-Service` at the command line and press *tab*. You would first see **Get-Service**, followed by the rest of the items in the previous screenshot as you hit *tab*. Tab completion is a huge benefit for your scripting productivity for several reasons, such as:

- You get the suggestions where you need them
- The suggestions are all valid command names
- The suggestions are consistently capitalized

In addition to being able to complete the command names, tab completion also gives suggestions for parameter names, property, and method names, and in PowerShell 3.0 and above, it gives the parameter values. The combination of tab completion improvements and IntelliSense in the ISE is so impressive that I recommend that people upgrade their development workstation to at least PowerShell 3.0, even if they are developing PowerShell code that will run in a 2.0 environment.

How does that work?

Knowing what commands you can execute is a big step, but it doesn't help much if you don't know how you can use them. Again, PowerShell is here to help you. To see a quick hint of how to use a cmdlet, write the cmdlet name followed by `-?`. The beginning of this output for `Get-Service` is shown as follows:



The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with icons for New, Open, Save, and Run is visible. The main area contains the following text:

```
PS C:\Users\Mike> Get-Service -?

NAME
  Get-Service

SYNOPSIS
  Gets the services on a local or remote computer.

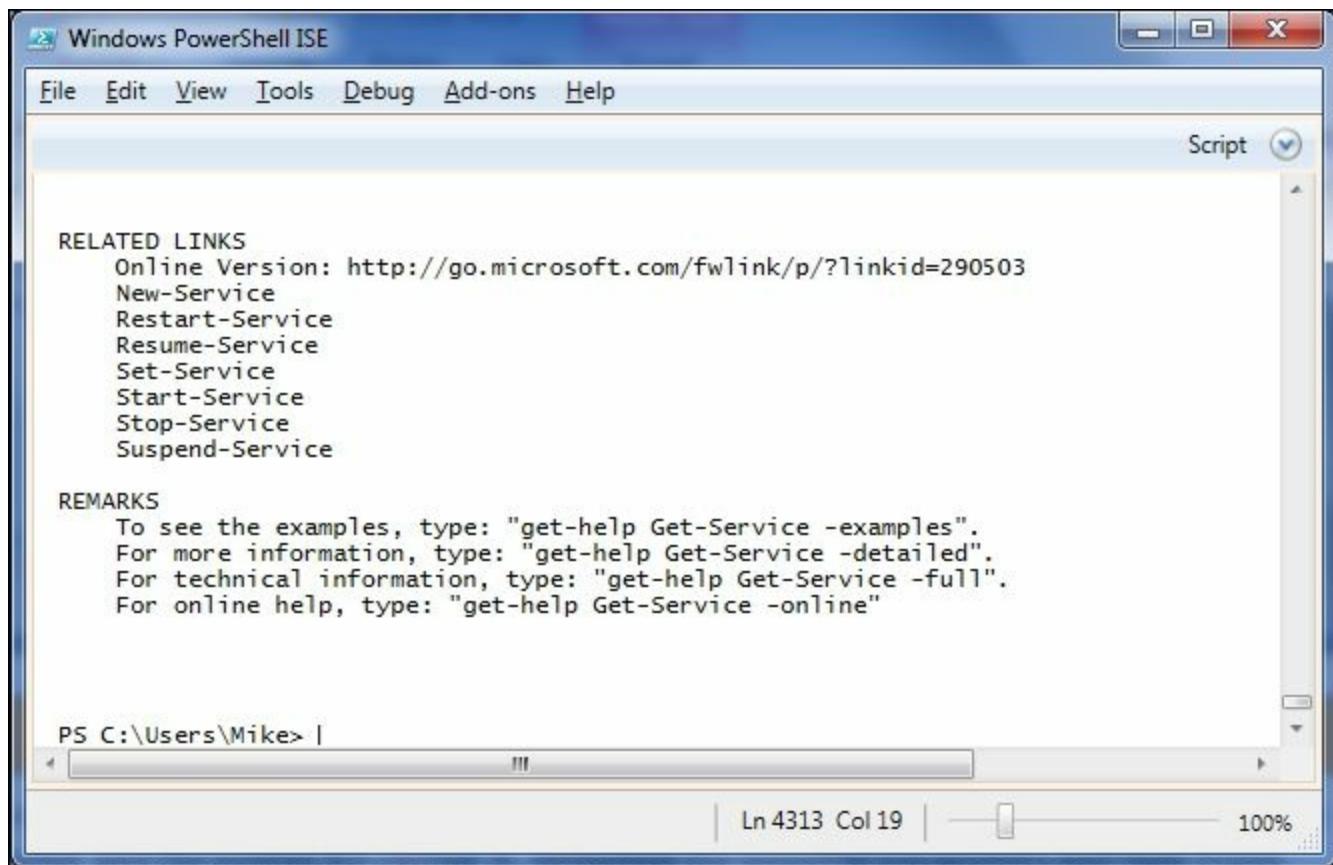
SYNTAX
  Get-Service [[-Name] <String[]>] [-ComputerName <String[]>]
  [-DependentServices] [-Exclude <String[]>] [-Include <String[]>]
  [-RequiredServices] [<CommonParameters>]

  Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude
  <String[]>] [-Include <String[]>] [-RequiredServices] -DisplayName
  <String[]> [<CommonParameters>]

  Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude
  <String[]>] [-Include <String[]>] [-InputObject <ServiceController[]>]
  [-RequiredServices] [<CommonParameters>]
```

The status bar at the bottom shows "Ln 4313 Col 19" and "100%".

Even this brief help, which was truncated to fit on the page, shows a brief synopsis of the cmdlet and the syntax to call it, including the possible parameters and their types. The rest of the display shows a longer description of the cmdlet, a list of the related topics, and some instructions about getting more help about `Get-Service`:



In the **Remarks** section, we can see that there's a cmdlet called `Get-Help` (the second of the "big 3" cmdlets) that allows us to view more extensive help in PowerShell. The first type of extra help we can see is the examples. The example output begins with the name and synopsis of the cmdlet and is followed, in the case of `Get-Service`, by 11 examples that range from simple to complex. The help for each cmdlet is different, but in general you will find these examples to be a treasure trove providing an insight into not only how the cmdlets behave in isolation, but also in combination with other commands in real scenarios:

The screenshot shows a Windows PowerShell ISE window. The title bar reads "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar with a "Script" button is visible. The main pane displays help for the "Get-Service" cmdlet. It starts with the command "PS C:\Users\Mike> get-help Get-Service -Examples". The help output includes the NAME section ("Get-Service") and the SYNOPSIS section ("Gets the services on a local or remote computer"). Below this is "EXAMPLE 1" showing the command "PS C:\>get-service". A note explains that this retrieves all services. "EXAMPLE 2" shows the command "PS C:\>get-service wmi*". A note specifies that this retrieves services starting with "WMI". The status bar at the bottom shows "Ln 221 Col 19" and "100%".

```
PS C:\Users\Mike> get-help Get-Service -Examples

NAME
  Get-Service

SYNOPSIS
  Gets the services on a local or remote computer.

----- EXAMPLE 1 -----

PS C:\>get-service

This command retrieves all of the services on the system. It behaves as
though you typed "get-service *". The default display shows the status,
service name, and display name of each service.

----- EXAMPLE 2 -----


PS C:\>get-service wmi*

This command retrieves services with service names that begin with "WMI"
!!!
```

Also, mentioned in this are the ways to display more information about the cmdlet using the `-Detailed` or `-Full` switches with `Get-Help`. The `-Detailed` switch shows the examples as well as the basic descriptions of each parameter. The `-Full` switch adds sections on inputs, outputs, and detailed parameter information to the `-Detailed` output.

Tip

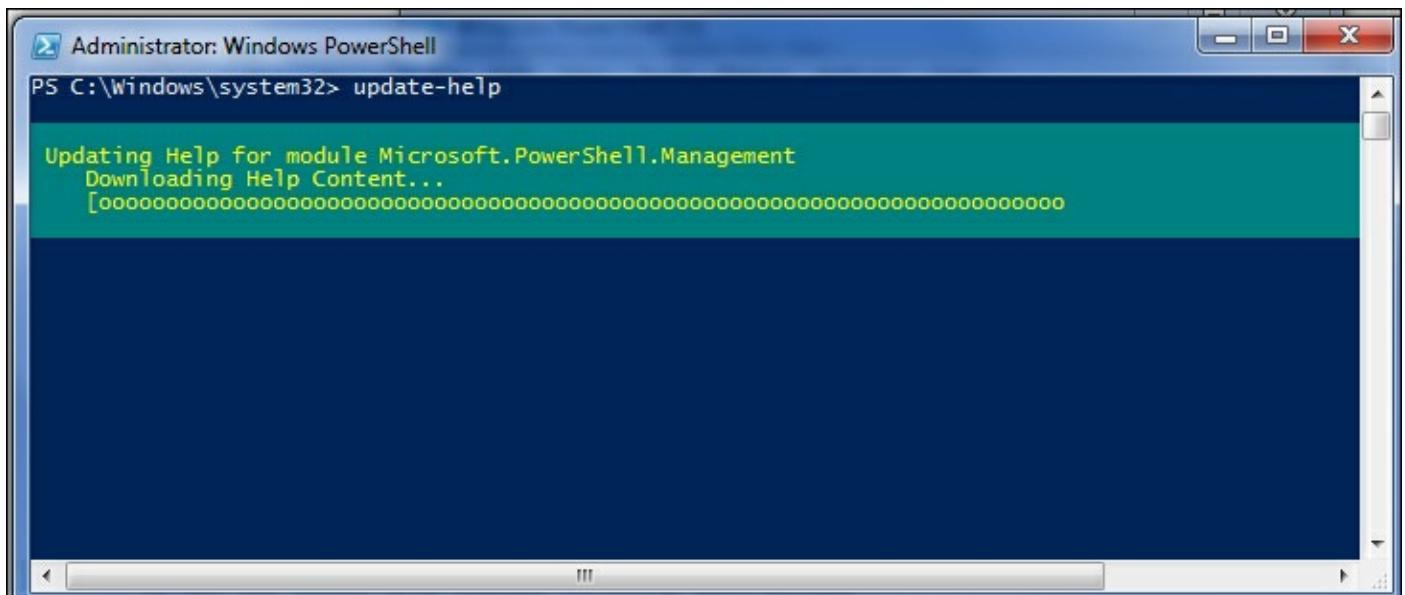
Using `Get-Help` for a cmdlet without specifying any other parameters gives the same output as using `-?` after the cmdlet name.

If you're running PowerShell 3.0 or above, instead of getting a complete help entry, you probably received a message like this:

REMARKS

To get the latest Help content including descriptions and examples type: Update-Help.

This is because in PowerShell 3.0, the PowerShell team switched to the concept of update-able help. In order to deal with time constraints around shipping and the complexity of releasing updates to installed software, the help content for the PowerShell modules is now updated via a cmdlet called `Update-Help`. Using this mechanism, the team can revise, expand, and correct the help content on a continual basis, and users can be sure to have the most recent content at all times:



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `PS C:\Windows\system32> update-help` is entered. The output shows the process of updating help content for the module `Microsoft.PowerShell.Management`, with a progress bar indicating the download of help content.

```
Administrator: Windows PowerShell
PS C:\Windows\system32> update-help

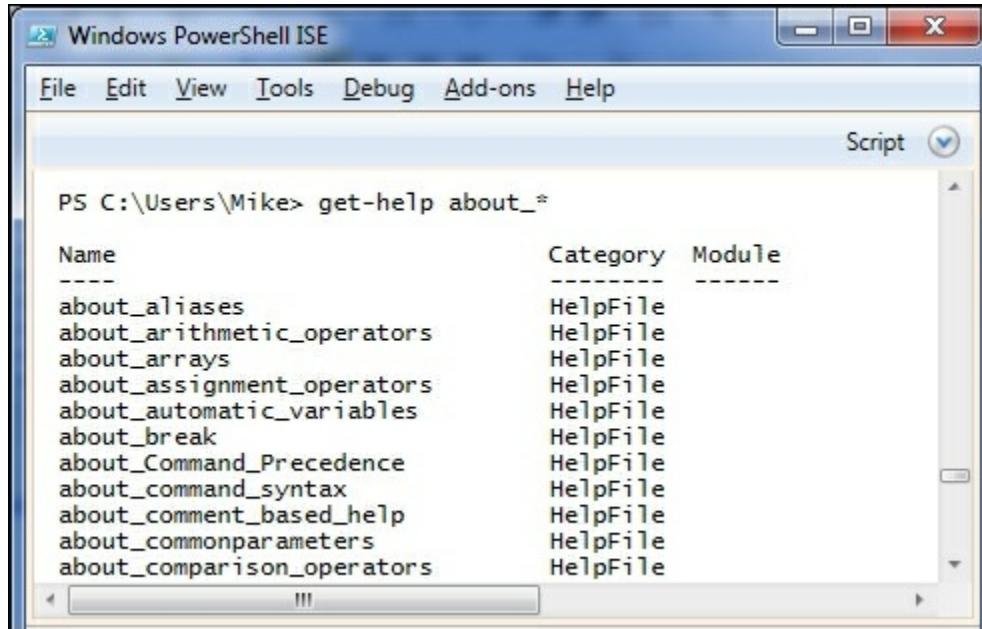
Updating Help for module Microsoft.PowerShell.Management
Downloading Help Content...
[oooooooooooooooooooooooooooooooooooooooooooooooooooo]
```

Tip

`Update-Help` requires an elevated session, so make sure that you start the PowerShell session as an administrator before trying to update your help content.

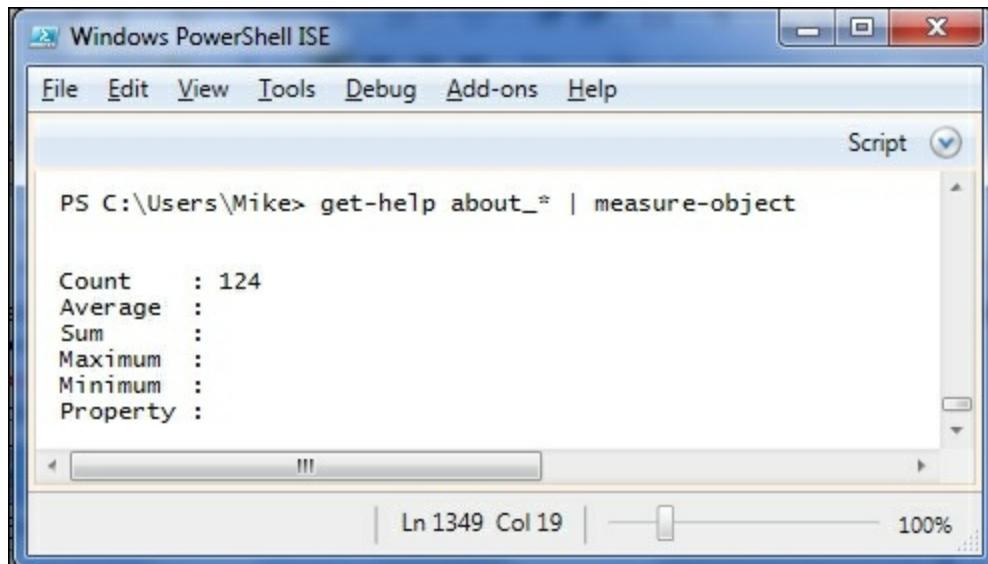
In addition to the help with individual cmdlets, PowerShell includes help content about all aspects of the PowerShell environment. These help topics are named beginning with `about_` and can also be viewed with the

Get-Help cmdlet. The list of topics can be retrieved using get-Help about_*:



Name	Category	Module
about_aliases	HelpFile	
about_arithmetic_operators	HelpFile	
about_arrays	HelpFile	
about_assignment_operators	HelpFile	
about_automatic_variables	HelpFile	
about_break	HelpFile	
about_Command_Precedence	HelpFile	
about_command_syntax	HelpFile	
about_comment_based_help	HelpFile	
about_commonparameters	HelpFile	
about_comparison_operators	HelpFile	

Using measure-object, as we saw in a previous section, we can see that there are 124 topics listed in my installation:

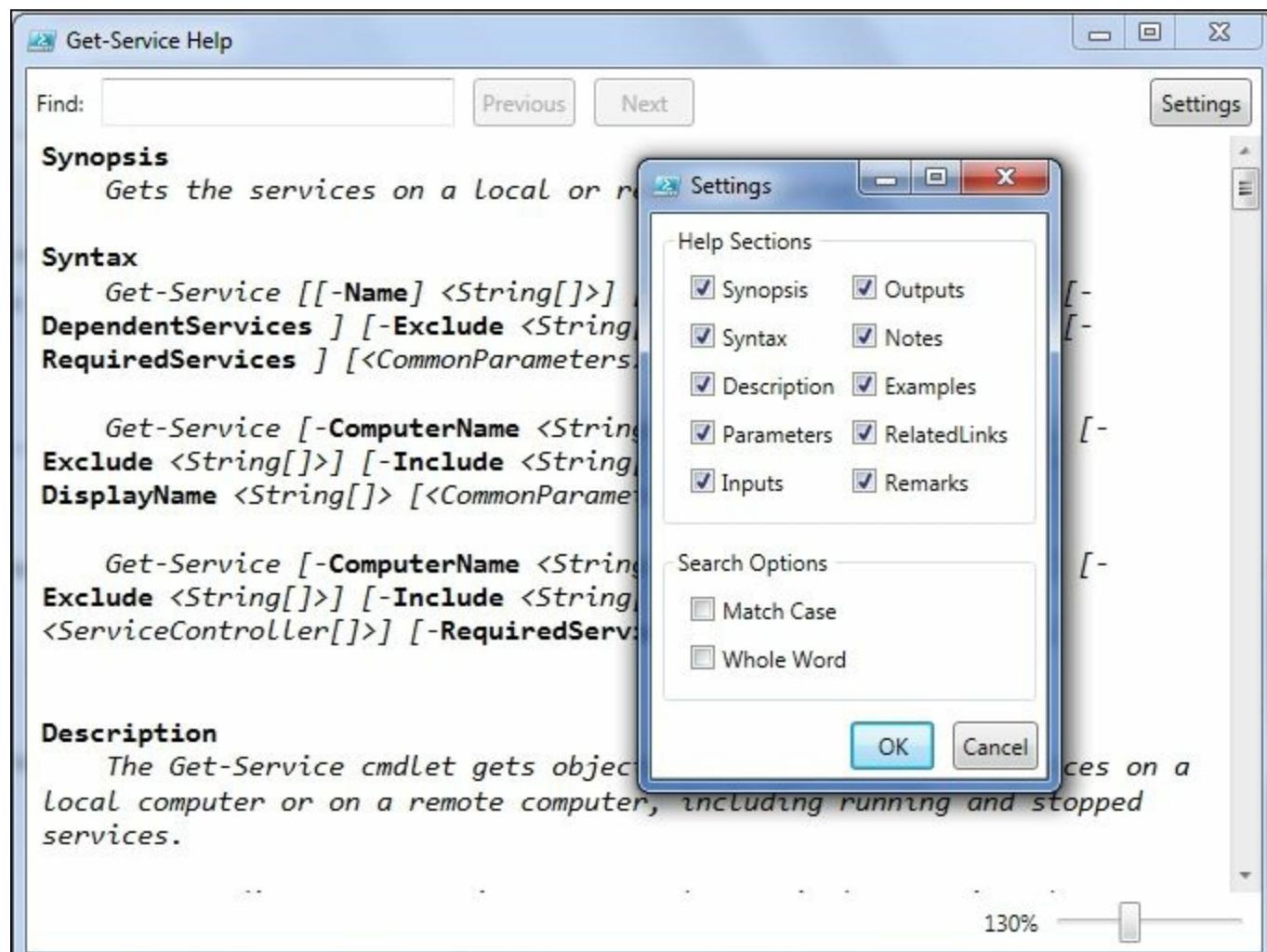


```
PS C:\Users\Mike> get-help about_* | measure-object

Count : 124
Average :
Sum :
Maximum :
Minimum :
Property :
```

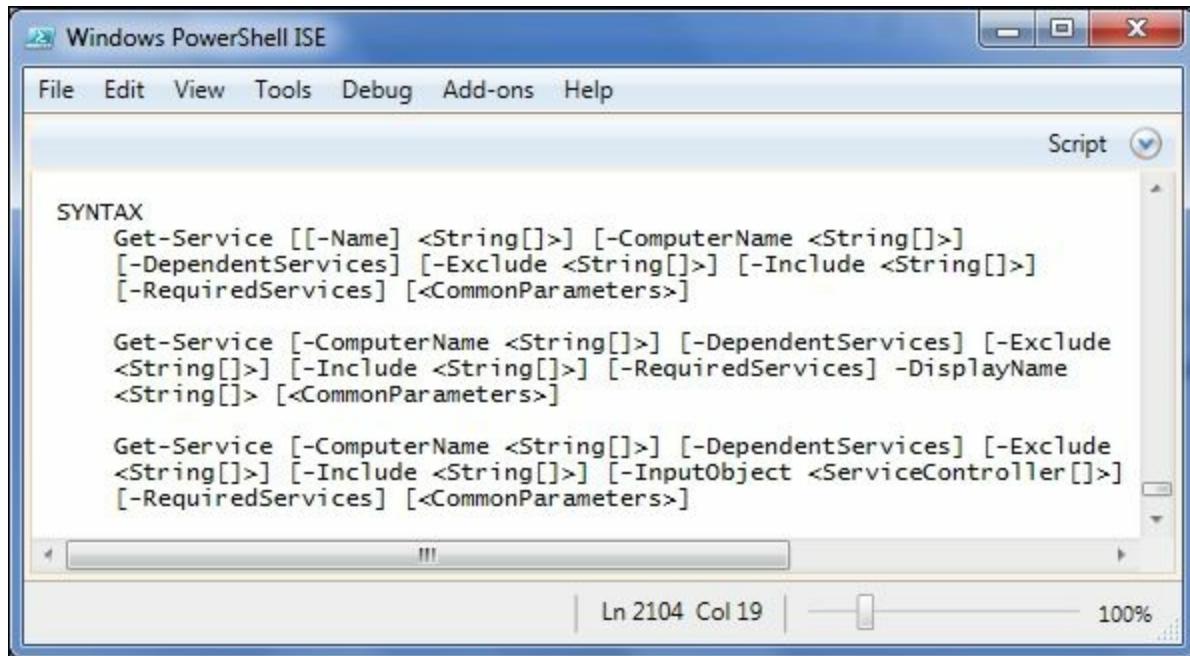
These help topics are often extremely long and contain some of the best documentation on the working of PowerShell you will ever find. For that reason, I will end each chapter in Module 1 with a list of help topics to read about the topics covered in the chapter.

In addition to reading the help content in the output area of the ISE and the text console, PowerShell 3.0 added a `-ShowWindow` switch to get-help that allows viewing in a separate window. All help content is shown in the window, but sections can be hidden using the settings button:



Interpreting the cmdlet syntax

The syntax section of the cmdlet help can be overwhelming at first, so let's drill into it a bit and try to understand it in detail. I will use the `get-service` cmdlet for an example, but the principles are same for any cmdlet:



A screenshot of the Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar has a "Script" button. The main area is titled "SYNTAX" and contains three examples of the Get-Service cmdlet with different parameter sets:

```
Get-Service [[-Name] <String[]>] [-ComputerName <String[]>]
[-DependentServices] [-Exclude <String[]>] [-Include <String[]>]
[-RequiredServices] [<CommonParameters>]

Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude
<String[]>] [-Include <String[]>] [-RequiredServices] -DisplayName
<String[]> [<CommonParameters>]

Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude
<String[]>] [-Include <String[]>] [-InputObject <ServiceController[]>]
[-RequiredServices] [<CommonParameters>]
```

The status bar at the bottom shows "Ln 2104 Col 19" and "100%".

The first thing to notice is that there are three different `get-service` calls illustrated here. They correspond to the PowerShell concept of **ParameterSets**, but you can think of them as different use cases for the cmdlet. Each ParameterSet, or use case will have at least one unique parameter. In this case, the first includes the `-Name` parameter, the second includes `-DisplayName`, and the third has the `-InputObject` parameter.

Each ParameterSet lists the parameters that can be used in a particular scenario. The way the parameter is shown in the listing, tells you how the parameter can be used. For instance, `[[-Name] <String[]>]` means that the name parameter has the following attributes:

- It is optional (because the whole definition is in brackets)
- The parameter name can be omitted (because the name is in

brackets)

- The parameter values can be an array (list) of strings (String[])

Tip

Many cmdlet parameters allow lists to be passed, rather than only a single value. Taking the time to check the types of parameters can make it simpler and quicker to write scripts, if you can eliminate looping because you are able to pass all the values at once.

In general, a parameter will be shown in one of the following forms:

Form	Meaning
[[-Param] T]	Optional parameter of type T with the name optional
[-Param T]	Optional parameter of type T with the name required
-Param T	Required parameter of type T
[-Param]	A switch (flag)

Applying this to the `Get-Service` syntax shown previously, we see that some valid calls of the cmdlet could be:

- `Get-Service WinRM`
- `Get-Service -Name WinRM`
- `Get-Service *SQL* -Exclude SQLWriter`
- `Get-Service aspnet_state,W3SVC`

Tip

Your turn!

Read the help for another cmdlet (`get-process` is a good candidate) and work through the syntax until you understand how the parameters are specified.

Summary

This chapter dealt with the first two of the "big three" cmdlets for learning PowerShell, `get-Command` and `get-Help`. These two cmdlets allow you to find out which commands are available and how to use them. In the next chapter, we will finish the "big 3" with the `get-member` cmdlet that will help us to figure out what to do with the output we receive.

For further reading

- `Get-Help Get-Command`
- `Get-Help Get-Help`
- `Get-Help Get-Member`
- `Get-Help about_Updateable_Help`
- `Get-Help Update-Help`
- `Get-Help about_Command_Syntax`

Chapter 3. Objects and PowerShell

In this chapter, we will learn about objects and how they relate to the PowerShell output. The specific topics in this chapter include the following:

- What are objects?
- Comparing DOS and PowerShell output
- The `Get-Member` cmdlet

Objects all the way down

One major difference between PowerShell and other command environments is that, in PowerShell, everything is an object. One result of this is that the output from the PowerShell cmdlets is always in the form of objects. Before we look at how this affects PowerShell, let's take some time to understand what we mean when we talk about objects.

Digging into objects

If everything is an object, it's probably worth taking a few minutes to talk about what this means. We don't have to be experts in object-oriented programming to work in PowerShell, but a knowledge of a few things is necessary.

In a nutshell, object-oriented programming involves encapsulating the related values and functionality in objects. For instance, instead of having variables for speed, height, and direction and a function called `PLOTPROJECTILE`, in object-oriented programming you might have a `Projectile` object that has the properties called `Speed`, `Height`, and `Direction` as well as a method called `Plot`. This `Projectile` object could be treated as a single unit and passed as a parameter to other functions. Keeping the values and the related code together has a certain sense to

it, and there are many other benefits to object-oriented programming.

Types, classes, and objects

The main concepts in object-oriented programming are types, classes, and objects. These concepts are all related and often confusing, but not difficult once you have them straight:

- A **type** is an abstract representation of structured data combining values (called properties), code (called methods), and behavior (called events). These attributes are collectively known as **members**.
- A **class** is a specific implementation of one or more types. As such, it will contain all the members that are specified in each of these types and possibly other members as well. In some languages, a class is also considered a type. This is the case in C#.
- An **object** is a specific instance of a class. Most of the time in PowerShell, we will be working with objects. As mentioned earlier, all output in PowerShell is in the form of objects.

PowerShell is build upon the .NET framework, so we will be referring to the .NET framework throughout Module 1. In the .NET framework, there is a type called `System.IO.FileSystemInfo`, which describes the items located in a filesystem (of course). Two classes that implement this type are `System.IO.FileInfo` and `System.IO.DirectoryInfo`. A reference to a specific file would be an object that was an instance of the `System.IO.FileInfo` class. The object would also be considered to be of the `System.IO.FileSystemInfo` and `System.IO.FileInfo` types.

Here are some other examples that should help make things clear:

- All the objects in the .NET ecosystem are of the `System.Object` type
- Many collections in .NET are of the `IEnumerable` type
- The `ArrayList` is of the `IList` and `ICollection` types among others

What are members?

Members are the attributes of types that are implemented in classes and instantiated in objects. Members come in many forms in the .NET framework, including properties, methods, and events.

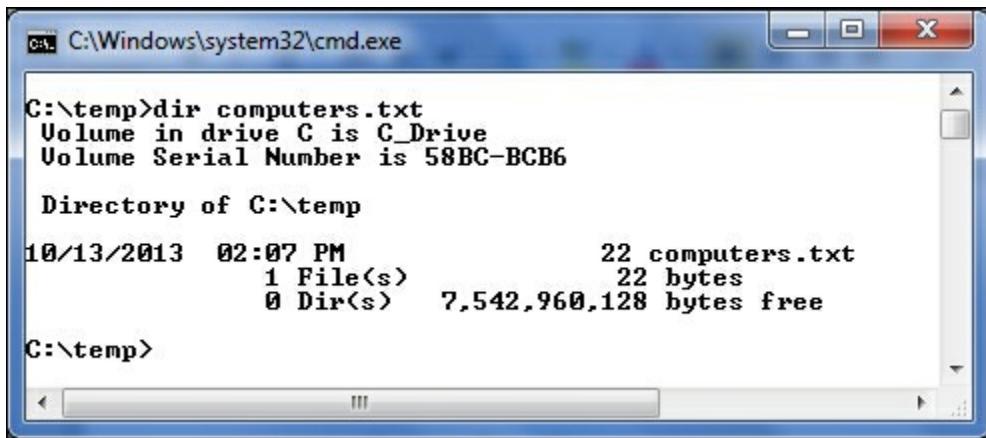
Properties represent the data contained in an object. For instance, in a `FileInfo` object (`System.IO.FileInfo`), there would be properties referring to the filename, file extension, length, and various `DateTime` values associated with the file. An object referring to a service (of the `System.ServiceProcess.ServiceController` type) would have properties for the display name of the service and the state (running or not).

Methods refer to the operations that the objects (or sometimes classes) can perform. A service object, for instance, can Start and Stop. A database command object can Execute. A file object can Delete or Encrypt itself. Most objects have some methods (for example, `ToString` and `Equals`) because they are of the `System.Object` type.

Events are a way for an object to notify that something has happened or that a condition has been met. Button objects, for instance, have an `OnClick` event that is triggered when they are clicked.

Now that we have an idea about objects, let's look back at a couple of familiar DOS commands and see how they deal with the output.

The DOS DIR command



C:\temp>dir computers.txt
Volume in drive C is C_Drive
Volume Serial Number is 58BC-BCB6

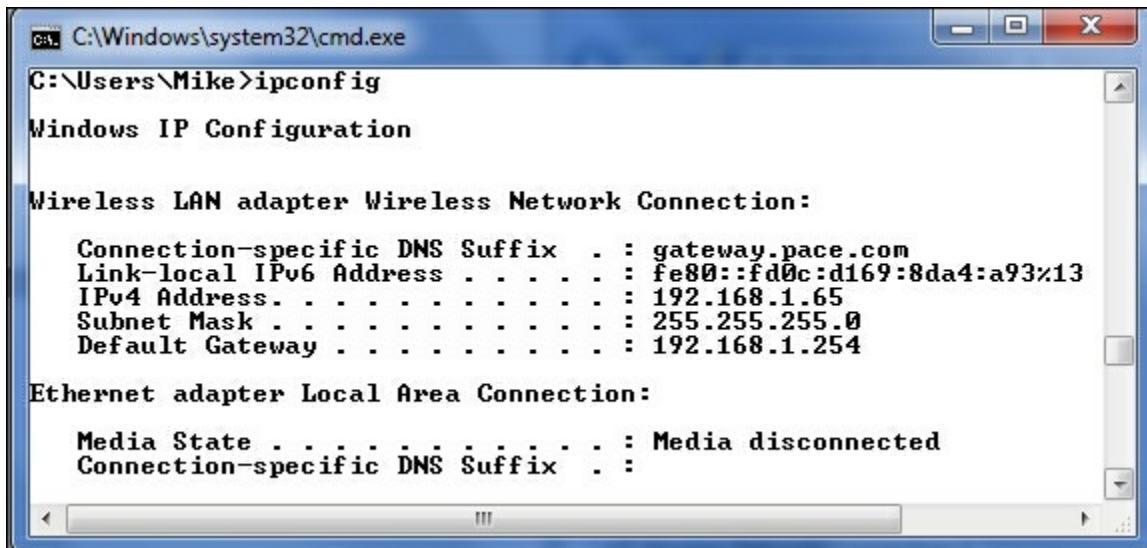
Directory of C:\temp
10/13/2013 02:07 PM 22 computers.txt
1 File(s) 22 bytes
0 Dir(s) 7,542,960,128 bytes free

C:\temp>

The output that we see in the preceding screenshot includes several details about a single file. Note that there is a lot of formatting included with the static text (for example, `Volume in drive`) and some tabular information, but the only way to get to these details is to understand exactly how the output is formatted and parse the output string accordingly. Think about how the output would change, if there were more than one file listed. If we included the `/S` switch, the output would have spanned over multiple directories, and would have been broken into sections accordingly. Finding a specific piece of information from this is not a trivial operation, and the process of retrieving the file length, for example, would be different from how you would go about retrieving the file extension.

The IPCONFIG command

Another command that we are all familiar with is the `IPConfig.exe` tool, which shows network adapter information. Here is the beginning of the output in my laptop:



The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The command 'ipconfig' was run, displaying network configuration details. The output includes sections for 'Wireless LAN adapter Wireless Network Connection' and 'Ethernet adapter Local Area Connection'. In the 'Wireless LAN adapter' section, properties like 'Connection-specific DNS Suffix', 'Link-local IPv6 Address', 'IPv4 Address', 'Subnet Mask', and 'Default Gateway' are listed with their corresponding values. The 'Ethernet adapter' section shows 'Media State' as 'Media disconnected' and 'Connection-specific DNS Suffix'.

```
C:\Users\Mike>ipconfig
Windows IP Configuration

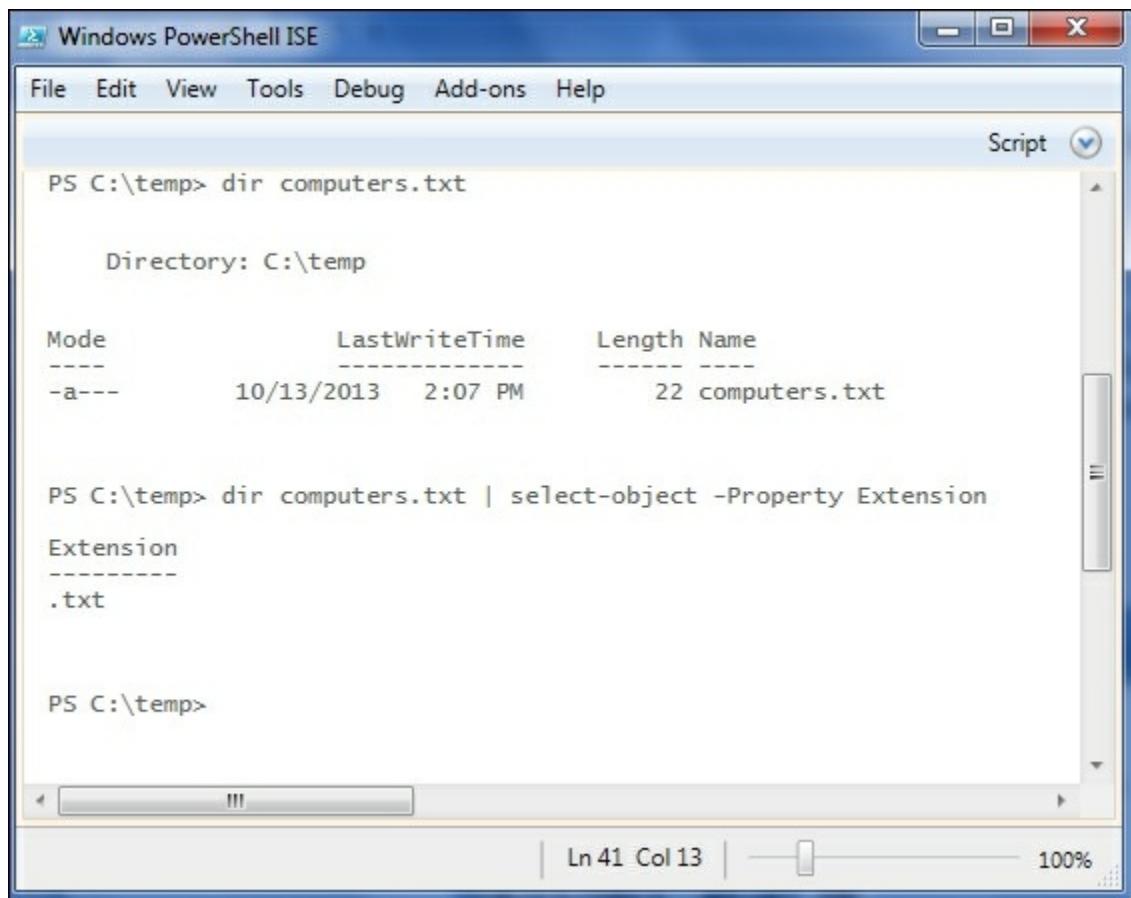
Wireless LAN adapter Wireless Network Connection:
  Connection-specific DNS Suffix . : gateway.pace.com
  Link-local IPv6 Address . . . . . : fe80::fd0c:d169:8da4:a93%13
  IPv4 Address . . . . . : 192.168.1.65
  Subnet Mask . . . . . : 255.255.255.0
  Default Gateway . . . . . : 192.168.1.254

Ethernet adapter Local Area Connection:
  Media State . . . . . : Media disconnected
  Connection-specific DNS Suffix . . :
```

Here, again, the output is very mixed. There is a lot of static text and a list of properties and values. The property names are very readable, which is nice for humans, but not so nice for computers trying to parse things out. The dot-leaders are again something that help to guide the eyes towards the property values, but will get in the way when we try to parse out the values that we are looking for. Since some of the values (such as IP addresses and subnet masks) already include dots, I imagine it will cause some confusion.

PowerShell for comparison

Let's look at some PowerShell commands and compare how easy it is to get what we need out of the output:



The screenshot shows a Windows PowerShell Integrated Scripting Environment (ISE) window titled "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar button labeled "Script" with a dropdown arrow is visible. The main pane displays PowerShell commands and their output:

```
PS C:\temp> dir computers.txt

Directory: C:\temp

Mode                LastWriteTime     Length Name
----                -----          ---- 
-a---   10/13/2013  2:07 PM        22    computers.txt

PS C:\temp> dir computers.txt | select-object -Property Extension

Extension
-----
.txt

PS C:\temp>
```

The status bar at the bottom indicates "Ln 41 Col 13" and "100%".

We can see that the output of the `dir` command looks similar to the output we saw in DOS, but we can follow this with `Select-Object` to pull out a specific property. This is possible because PowerShell commands always output objects. In this case, the object in question has a property called `Extension`, which we can inspect. We will talk about the `select-object` cmdlet in detail in the next chapter, but for now it is enough to know that it can be used to limit the output to a specific set of properties from the original objects.

The Get-Member cmdlet

One way to find the members of a class is to look up this class online in the **MicroSoft Developers Network (MSDN)**. For instance, the `FileInfo` class is found at <https://msdn.microsoft.com/en-us/library/system.io.fileinfo>.

Although this is a good reference, it's not very handy to switch back and forth between PowerShell and a browser to look at the classes all the time. Fortunately, PowerShell has a very handy way to give this information, the `Get-Member` cmdlet. This is the third of the "big 3" cmdlets, following `Get-Command` and `Get-Help`.

The most common way to use the `Get-Member` cmdlet is to **pipe** data into it. Piping is a way to pass data from one cmdlet to another, and is covered in depth in the next chapter. Using a pipe with `Get-Member` looks like this:

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script

```
PS C:\Windows\system32> Get-Service | Get-Member
```

TypeName: System.ServiceProcess.ServiceController

Name	MemberType	Definition
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDependedOn
Disposed	Event	System.EventHandler Disposed(System....)
Close	Method	void Close()
Continue	Method	void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef Creat...
Dispose	Method	void Dispose(), void IDisposable.Dis...
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 InitializeLifetimeServ...
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.ServicePro...
CanPauseAndContinue	Property	bool CanPauseAndContinue {get;}
CanShutdown	Property	bool CanShutdown {get;}
CanStop	Property	bool CanStop {get;}
Container	Property	System.ComponentModel.IContainer Con...
DependentServices	Property	System.ServiceProcess.ServiceControl...
DisplayName	Property	string DisplayName {get;set;}
MachineName	Property	string MachineName {get;set;}
ServiceHandle	Property	System.Runtime.InteropServices.SafeH...
ServiceName	Property	string ServiceName {get;set;}
ServicesDependedOn	Property	System.ServiceProcess.ServiceControl...
ServiceType	Property	System.ServiceProcess.ServiceType Se...
Site	Property	System.ComponentModel.ISite Site {ge...
Status	Property	System.ServiceProcess.ServiceControl...
ToString	ScriptMethod	System.Object ToString();

Ln 43 Col 25 | 100%

The `Get-Member` cmdlet looks at all the objects in its input, and provides output for each distinct class. In this output, we can see that `Get-Service` only outputs a single type,

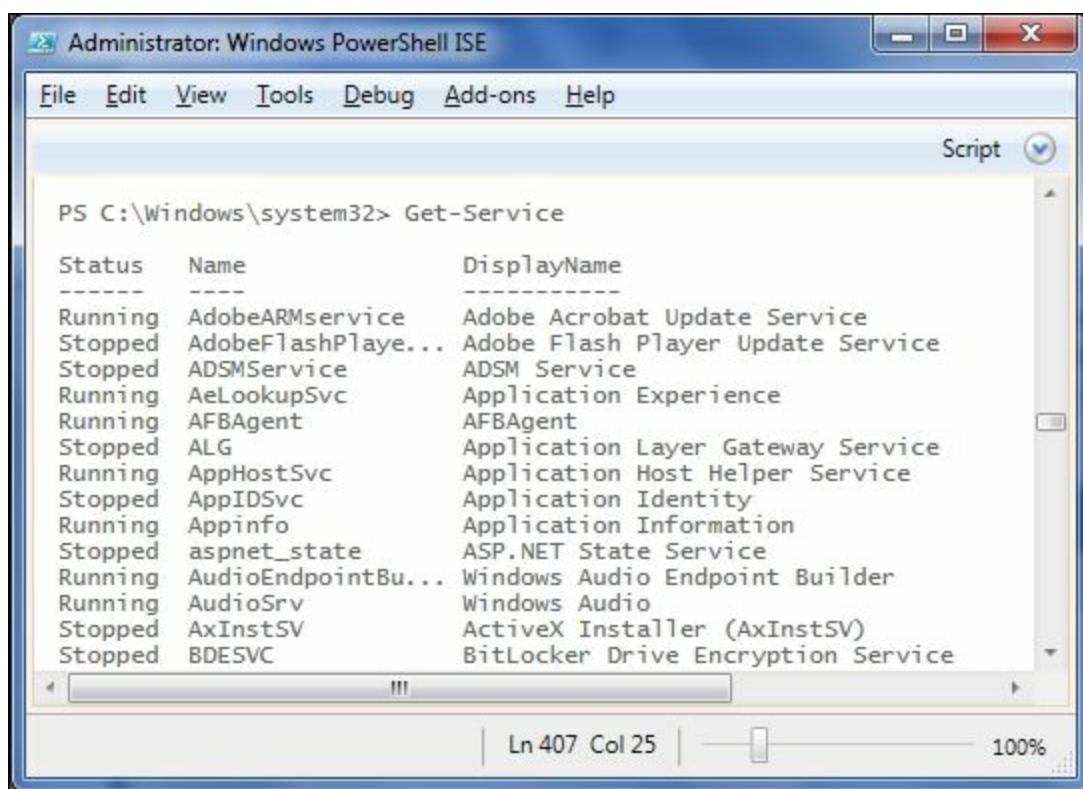
`System.ServiceProcess.ServiceController`. The name of the class is followed by the list of members, type of each member as well as definition for the member. The member definition shows the type of properties, signature for events, and methods, which includes the return type as well as the types of parameters.

Tip

Your turn!

Use `Get-Member` to see the classes output by the `Dir` (or `Get-ChildItem`) cmdlet. Note that there are two different classes listed.

One thing that can be confusing is that `get-member` usually shows more properties than those shown in the output. For instance, the `Get-Member` output for the previous `Get-Service` cmdlet shows 13 properties but the output of `Get-Service` displays only three.



The screenshot shows a Windows PowerShell ISE window titled "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu shows "Script". The command "PS C:\Windows\system32> Get-Service" is run, displaying a table of service status, name, and display name. The table has columns for Status, Name, and DisplayName. The services listed include AdobeARMservice, AdobeFlashPlaye..., ADSMService, AeLookupSvc, AFBAgent, ALG, AppHostSvc, AppIDSvc, Appinfo, aspnet_state, AudioEndpointBu..., AudioSrv, AxInstSV, and BDESVC. The display names provide a brief description of each service's function.

Status	Name	DisplayName
Running	AdobeARMservice	Adobe Acrobat Update Service
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service
Stopped	ADSMService	ADSM Service
Running	AeLookupSvc	Application Experience
Running	AFBAgent	AFBAgent
Stopped	ALG	Application Layer Gateway Service
Running	AppHostSvc	Application Host Helper Service
Stopped	AppIDSvc	Application Identity
Running	Appinfo	Application Information
Stopped	aspnet_state	ASP.NET State Service
Running	AudioEndpointBu...	Windows Audio Endpoint Builder
Running	AudioSrv	Windows Audio
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Stopped	BDESVC	BitLocker Drive Encryption Service

The reason for this is that PowerShell has a powerful formatting system that is configured with some default formats for familiar objects. Rest assured that all the properties are there. A couple of quick ways to see all the properties are to use the `Select-Object` cmdlet we saw earlier in this chapter or to use the `Format-List` cmdlet and force all the properties to be shown. Here, we use `Select-Object` and a wildcard to specify all the properties:

The screenshot shows a Windows PowerShell Integrated Scripting Environment (ISE) window titled "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar at the top right has a "Script" button with a dropdown arrow. The main pane displays the command "PS C:\temp> Get-Service Appinfo | Select-Object -Property *" followed by its output:

```
PS C:\temp> Get-Service Appinfo | Select-Object -Property *
Name          : Appinfo
RequiredServices : {RpcSs, ProfSvc}
CanPauseAndContinue : False
CanShutdown      : False
CanStop         : True
DisplayName     : Application Information
DependentServices : {}
MachineName    :
ServiceName    : Appinfo
ServicesDependedOn : {RpcSs, ProfSvc}
ServiceHandle   : SafeServiceHandle
Status          : Running
ServiceType     : Win32ShareProcess
Site            :
Container       :
```

The bottom status bar indicates "Ln 25 Col 13" and "100%".

The `Format-List` cmdlet with a wildcard for properties gives output that looks the same. We will discuss how the output of these two cmdlets actually differs, as well as elaborate on PowerShell's formatting system in [Chapter 5, *Formatting Output*](#).

Where did these all come from?

Technet24.ir

If you look closely at the `ServiceController` members listed in the previous figure, you will notice a few members that aren't strictly properties, methods, or events. PowerShell has a mechanism called the Extended Type System that allows PowerShell to add members to classes or to individual objects. In the case of the `SystemController` objects, PowerShell adds a `Name` alias for the `ServiceName` property and a `RequiredServices` alias for the built-in `ServicesDependedOn` property.

Tip

Your turn!

Use `Get-Member` with `Dir` and `Get-Process` and see what members PowerShell has added. Look these classes up on MSDN and verify that those properties aren't delivered as part of the .NET framework.

Summary

In this chapter, we discussed the importance of objects as output from the PowerShell cmdlets. After a brief primer on types, classes, and objects, we spent some time getting used to the `Get-Member` cmdlet. In the next chapter, we will cover the PowerShell pipeline and common pipeline cmdlets.

For further reading

- Get-Help about_objects
- Get-Help about_properties
- Get-Help about_methods
- Get-Help about_events
- Get-Help Get-Member

Chapter 4. Life on the Assembly Line

The object-oriented pipeline is one of the distinctive features of the PowerShell environment. The ability to refer to properties of arbitrary objects without parsing increases the expressiveness of the language and allows you to work with all kinds of objects with ease.

In this chapter, we will cover the following topics:

- How the pipeline works
- Some of the most common cmdlets to deal with data on the pipeline:
 - Sort-Object
 - Where-Object
 - Select-Object
 - Group-Object

The pipeline as an assembly line

The pipeline in PowerShell is a mechanism to get data from one command to another. Simply put, the data that is output from the first command is treated as input to the next command in the pipeline. The pipeline isn't limited to two commands, though. It can be extended practically as long as you like, although readability would suffer if the line got too long.

Here is a simple pipeline example:

The screenshot shows a Windows PowerShell ISE window. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar button labeled 'Script' with a checkmark is visible. The main pane displays the following PowerShell command:

```
PS C:\Users\Mike> get-service | where Status -eq Running | select Name,DisplayName -first 5
```

The output shows a table with two columns: 'Name' and 'DisplayName'. The 'Name' column lists service names like AdobeARMservice, AFBAgent, AppHostSvc, Appinfo, and AudioEndpointBuilder. The 'DisplayName' column provides more descriptive names for these services.

Name	DisplayName
AdobeARMservice	Adobe Acrobat Update Service
AFBAgent	AFBAgent
AppHostSvc	Application Host Helper Service
Appinfo	Application Information
AudioEndpointBuilder	Windows Audio Endpoint Builder

At the bottom of the window, status information indicates 'Ln 1660 Col 92' and a zoom level of '100%'. The overall interface is a standard Windows-style application window.

Tip

In this example, I've used some common aliases for cmdlets (`where`, `select`) to keep the line from wrapping. I'll try to include aliases when I mention cmdlets, but if you can't figure out what a command is referring to, remember you can always use `Get-Command` to find out what is going on. For example, `Get-Command where` tells you `where` is an alias for `Where-Object`. In this case, `select` is an alias for `Select-Object`.

The execution of this pipeline can be thought of in the following sequence:

- Get the list of services
- Choose the services that have the `Running` status
- Select the first five services
- Output the **Name** and **Display Name** of each one

Even though this is a single line, it shows some of the power of PowerShell. The line is very expressive, doesn't include a lot of extra syntactic baggage, and doesn't even require any variables. It also doesn't use explicit types or loops. It is a very unexceptional bit of PowerShell code, but this single line represents logic that would take several lines of code in a traditional language to express.

This isn't your DOS or Linux pipeline

DOS and Linux (and Unix, for that matter) have had pipes for a long time. Pipes in these systems work similar to how PowerShell pipes work on one level. In all of them, output is streamed from one command to the next. In other shells, though, the data being passed is simple, flat text.

For a command to use the text, it either needs to parse the text to get to the interesting bits, or it needs to treat the entire output like a blob. Since Linux and Unix use text-based configurations for most operating system functions, this makes sense. A wide range of tools to parse and find substrings is available in these systems, and scripting can be very complex.

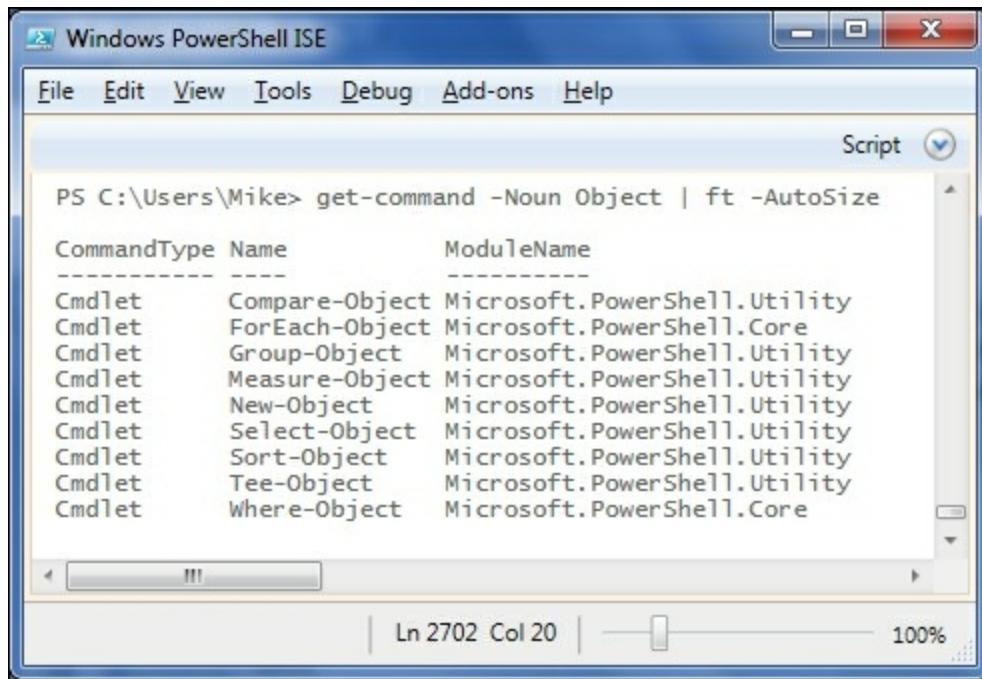
Objects at your disposal

In Windows, however, there aren't a lot of text-based configurations. Most components are managed using the Win32 or .NET APIs. PowerShell is built upon the .NET framework and leverages the .NET object model instead of using text as the primary focus. As data in the pipeline is always in the form objects, you rarely need to parse it and can directly deal with the properties of the objects themselves. As long as the properties are named reasonably (and they usually are), you will be able to quickly get to the data you need.

Dealing with pipeline data

Since commands on the pipeline have access to object properties, several general-purpose cmdlets exist to perform common operations. Since these cmdlets can work with any kind of object, they use `Object` as the noun (you remember the Verb-Noun naming convention for cmdlets, right?).

To find the list of these cmdlets, we can use the `Get-Command` cmdlet:



The screenshot shows a Windows PowerShell ISE window. The command `get-command -Noun Object | ft -AutoSize` is run in the top pane. The bottom pane shows the resulting table:

CommandType	Name	ModuleName
Cmdlet	Compare-Object	Microsoft.PowerShell.Utility
Cmdlet	ForEach-Object	Microsoft.PowerShell.Core
Cmdlet	Group-Object	Microsoft.PowerShell.Utility
Cmdlet	Measure-Object	Microsoft.PowerShell.Utility
Cmdlet	New-Object	Microsoft.PowerShell.Utility
Cmdlet	Select-Object	Microsoft.PowerShell.Utility
Cmdlet	Sort-Object	Microsoft.PowerShell.Utility
Cmdlet	Tee-Object	Microsoft.PowerShell.Utility
Cmdlet	Where-Object	Microsoft.PowerShell.Core

Tip

`ft` is an alias for the `Format-Table` cmdlet, which I'm using here to get the display to fit more nicely on the screen. It will be covered in depth in [Chapter 5, Formatting Output](#).

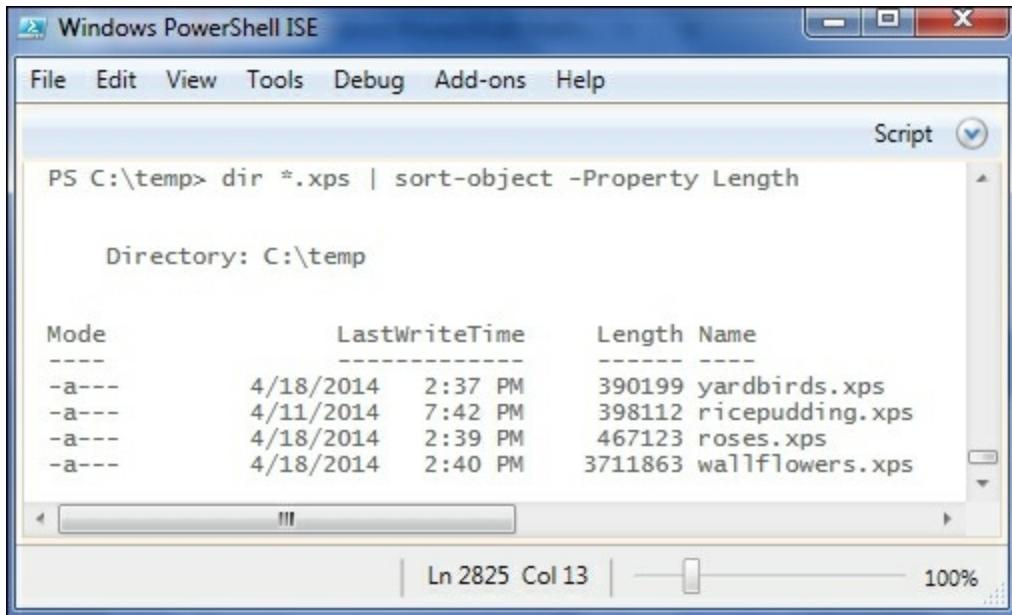
The `Sort-Object`, `Select-Object`, and `Where-Object` cmdlets are some of the most used cmdlets in PowerShell.

The `Sort-Object` cmdlet

Sorting data can be interesting. Have you ever sorted a list of numbers

only to find that 11 came between 1 and 2? That's because the sort that you used treated the numbers as text. Sorting dates with all the different culture-specific formatting can also be a challenge. Fortunately, PowerShell handles sorting details for you with the `Sort-Object` cmdlet.

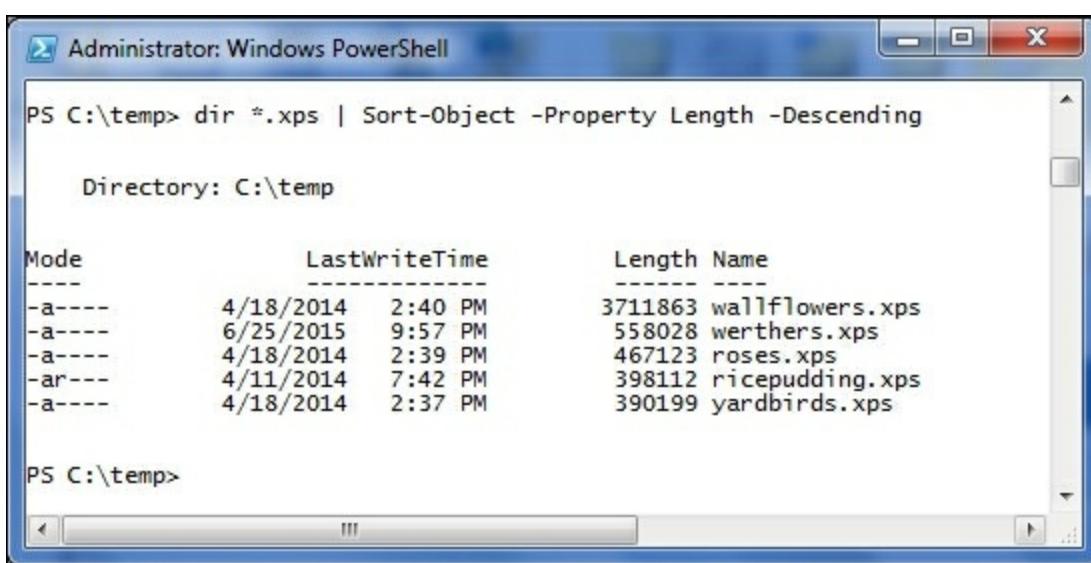
Let's look at a few examples of the `Sort-Object` cmdlet before getting into the details. We'll start by sorting a directory listing by length:



A screenshot of the Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu shows "Script". The command entered is "PS C:\temp> dir *.xps | sort-object -Property Length". The output shows a table with columns: Mode, LastWriteTime, Length, and Name. The files listed are yardbirds.xps, ricepudding.xps, roses.xps, and wallflowers.xps, ordered by length.

Mode	LastWriteTime	Length	Name
-a---	4/18/2014 2:37 PM	390199	yardbirds.xps
-a---	4/11/2014 7:42 PM	398112	ricepudding.xps
-a---	4/18/2014 2:39 PM	467123	roses.xps
-a---	4/18/2014 2:40 PM	3711863	wallflowers.xps

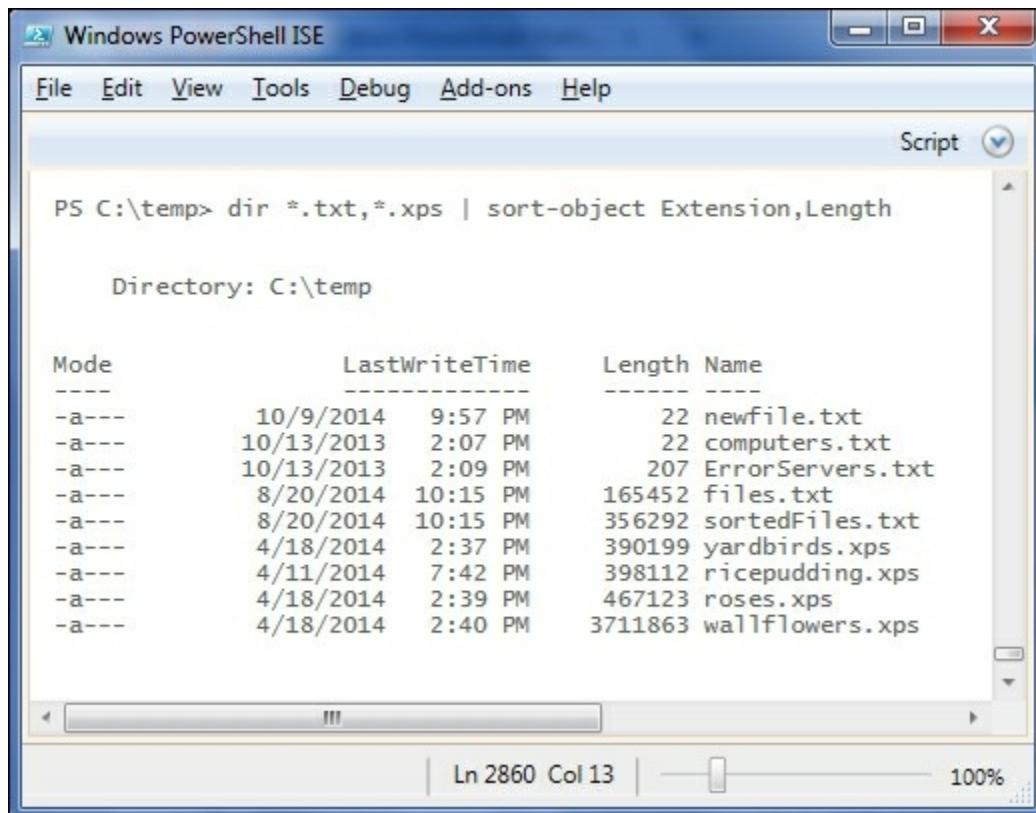
Sorting this in reverse isn't difficult either.



A screenshot of the Administrator: Windows PowerShell window. The title bar says "Administrator: Windows PowerShell". The command entered is "PS C:\temp> dir *.xps | Sort-Object -Property Length -Descending". The output shows a table with columns: Mode, LastWriteTime, Length, and Name. The files listed are wallflowers.xps, werthers.xps, roses.xps, ricepudding.xps, and yardbirds.xps, ordered by length in descending order.

Mode	LastWriteTime	Length	Name
-a---	4/18/2014 2:40 PM	3711863	wallflowers.xps
-a---	6/25/2015 9:57 PM	558028	werthers.xps
-a---	4/18/2014 2:39 PM	467123	roses.xps
-ar---	4/11/2014 7:42 PM	398112	ricepudding.xps
-a---	4/18/2014 2:37 PM	390199	yardbirds.xps

Sorting by more than one property is a breeze as well. Here, I omitted the parameter name (-Property) to shorten the command-line a bit:



A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The title bar says "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A dropdown menu shows "Script" selected. The main window displays a command prompt: "PS C:\temp> dir *.txt,*.xps | sort-object Extension,Length". Below the command, the output shows a list of files in the directory C:\temp, sorted by extension and length. The output is as follows:

Mode	LastWriteTime	Length	Name
-a---	10/9/2014 9:57 PM	22	newfile.txt
-a---	10/13/2013 2:07 PM	22	computers.txt
-a---	10/13/2013 2:09 PM	207	ErrorServers.txt
-a---	8/20/2014 10:15 PM	165452	files.txt
-a---	8/20/2014 10:15 PM	356292	sortedFiles.txt
-a---	4/18/2014 2:37 PM	390199	yardbirds.xps
-a---	4/11/2014 7:42 PM	398112	ricepudding.xps
-a---	4/18/2014 2:39 PM	467123	roses.xps
-a---	4/18/2014 2:40 PM	3711863	wallflowers.xps

Tip

You try it!

Use the `Sort-object` cmdlet in conjunction with `dir` (`Get-ChildItem`), `Get-Service`, or `Get-Process`. Try sorting by more than one property or in reverse. Note that the dates and numbers are sorted correctly, even though the text might not be alphabetically sorted.

Looking at the brief help for the `Sort-Object` cmdlet, we can see a few other parameters, such as:

- `-Unique` (return distinct items found in the input)
- `-CaseSensitive` (force a case-sensitive sort)
- `-Culture` (specify what culture to use while sorting)

As it turns out, you will probably find few occasions to use these parameters and will be fine with the `-Property` and `-Descending`

parameters.

The Where-Object cmdlet

Another cmdlet that is extremely useful is the **Where-Object** cmdlet. `Where-Object` is used to filter the pipeline data based on a condition that is tested for each item in the pipeline. Any object in the pipeline that causes the condition to evaluate to a true value is output from the `Where-Object` cmdlet. Objects that cause the condition to evaluate to a false value are not passed on as output.

For instance, we might want to find all the files that are below 100 bytes in size in the `c:\temp` directory. One way to do that is to use the simplified, or comparison syntax for `Where-Object`, which was introduced in PowerShell 3.0. In this syntax, the command would look like this:

```
Dir c:\temp | Where-Object Length -lt 100
```

In this syntax, we can compare a single property of each object with a constant value. The comparison operator here is `-lt`, which is how PowerShell expresses "less than".

Tip

All PowerShell comparison operators start with a dash. This can be confusing, but the `<` and `>` symbols have an entrenched meaning in shells, so they can't be used as operators. Common operators include `-eq`, `-ne`, `-lt`, `-gt`, `-le`, `-ge`, `-not`, and `-like`. For a full list of operators, try `get-help about_operators`.

If you need to use PowerShell 1.0 or 2.0, or need to specify a condition more complex than it is allowed in the simplified syntax, you can use the general or **scriptblock** syntax. Expressing the same condition using this form looks as follows:

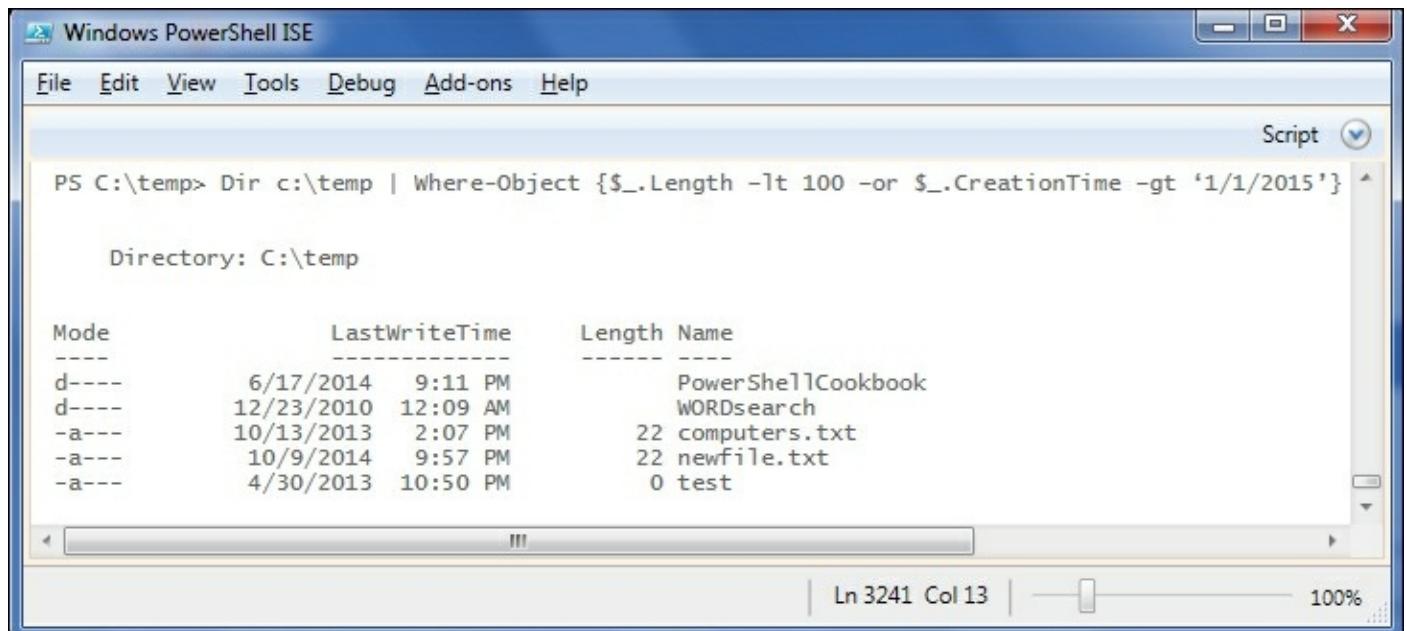
```
Dir c:\temp | Where-Object {$_ .Length -lt 100}
```

This looks a lot more complicated, but it's not so bad. The construction

in curly braces is called a scriptblock, and is simply a block of the PowerShell code. In the scriptblock syntax, `$_` stands for the current object in the pipeline and we're referencing the **Length** property of that object using dot-notation. The good thing about the general syntax of Where-Object is that we can do more inside the scriptblock than simply test one condition. For instance, we could check for files below 100 bytes or those that were created after 1/1/2015, as follows:

```
Dir c:\temp | Where-Object {$_ .Length -lt 100 -or  
$_ .CreationTime -gt '1/1/2015'}
```

Here's this command running on my laptop:



The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar button labeled "Script" is selected. The command entered in the console is:

```
PS C:\temp> Dir c:\temp | Where-Object {$_ .Length -lt 100 -or  
$_ .CreationTime -gt '1/1/2015'}
```

The output shows the directory contents of C:\temp:

Mode	LastWriteTime	Length	Name
d----	6/17/2014 9:11 PM		PowerShellCookbook
d----	12/23/2010 12:09 AM		WORDsearch
-a---	10/13/2013 2:07 PM	22	computers.txt
-a---	10/9/2014 9:57 PM	22	newfile.txt
-a---	4/30/2013 10:50 PM	0	test

If you're using PowerShell 3.0 or above and need to use the scriptblock syntax, you can substitute `$_` with `$PSItem` in the scriptblock. The meaning is the same and `$PSItem` is a bit more readable. It makes the line slightly longer, but it's a small sacrifice to make for readability's sake.

The examples that I've shown so far, have used simple comparisons with properties, but in the scriptblock syntax, any condition can be included. Also, note that any value other than a logical false value (expressed by `$false` in PowerShell), 0, or an empty string ("") is considered to be true

`($true)` in PowerShell. So, for example, we could filter objects using the `Get-Member` cmdlet to only show objects that have a particular property, as follows:

```
Dir | where-object {$_ | get-member Length}
```

This will return all objects in the current directory that have a `Length` property. Since files have lengths and directories don't, this is one way to get a list of files and omit the subdirectories.

Tip

You try it!

Use the `Where-Object` cmdlet to find all of the `*.ps*` files in `$PSHOME` that are larger than 1 kilobyte in size. Remember that you can express 1kilobyte using the KB unit suffix (1KB).

The Select-Object cmdlet

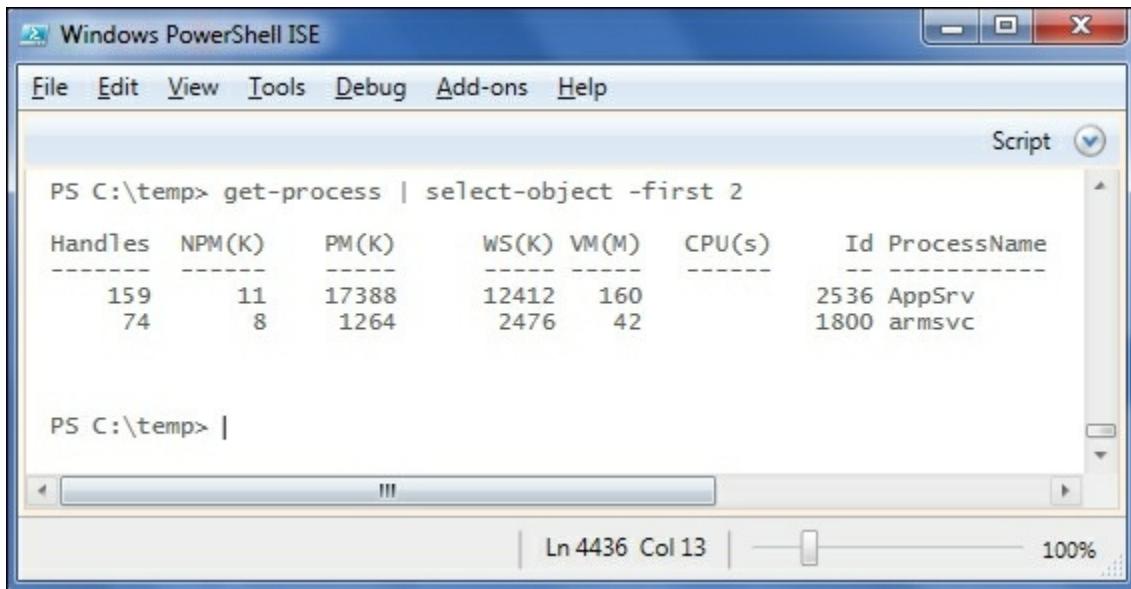
The **Select-Object** cmdlet is a versatile cmdlet that you will find yourself using often. There are three main ways that it is used:

- Limiting the number of the objects returned
- Limiting the properties of the objects returned
- Retrieving the value of a single property of the objects in the pipeline

Limiting the number of objects returned

Sometimes, you just want to see a few of the objects that are in the pipeline. To accomplish this, you can use the `-First`, `-Last`, and `-Skip` parameters. The `-First` parameter indicates that you want to see a particular number of objects from the beginning of the list of objects in the pipeline. Similarly, the `-Last` parameter selects objects from the end of the list of objects in the pipeline.

For instance, getting the first two processes in the list from `Get-Process` is simple:



The screenshot shows a Windows PowerShell ISE window. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar button labeled "Script" with a dropdown arrow is visible. The main pane displays the command "PS C:\temp> get-process | select-object -first 2" followed by its output. The output is a table with columns: Handles, NPM(K), PM(K), WS(K), VM(M), CPU(s), Id, and ProcessName. Two rows of data are shown:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
159	11	17388	12412	160		2536	AppSrv
74	8	1264	2476	42		1800	armsvc

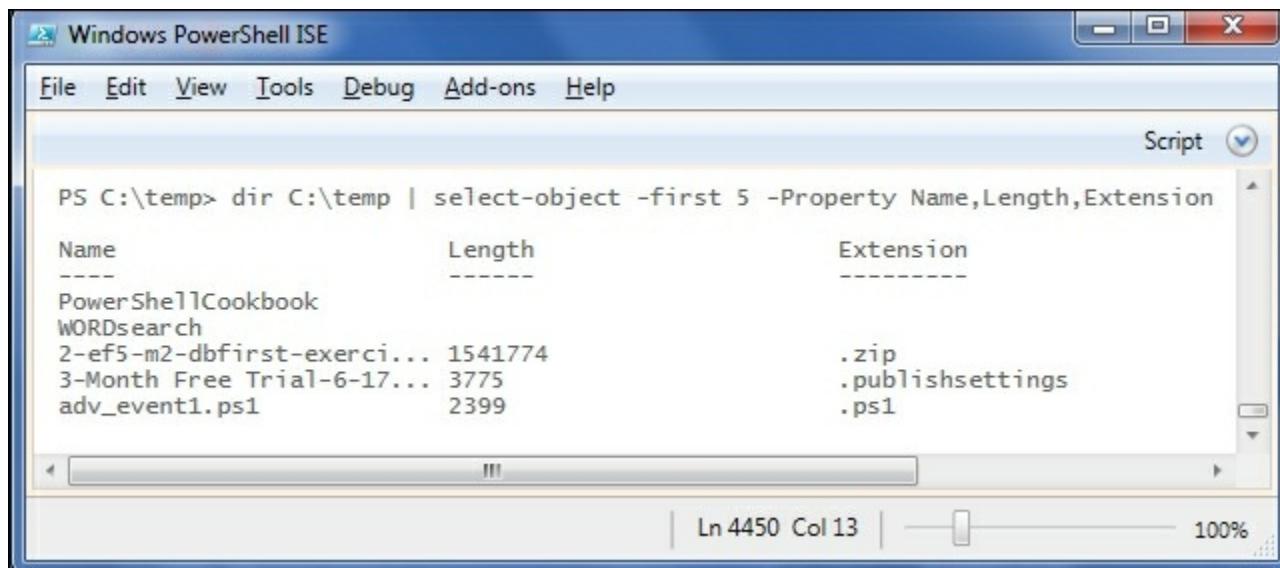
The status bar at the bottom indicates "Ln 4436 Col 13" and "100%".

Since we didn't use `Sort-Object` to force the order of the objects in the pipeline, we don't know that these are the first alphabetically, but they were the first two that were output from `Get-Process`.

You can use `-Skip` to cause a certain number of objects to be bypassed before returning the objects. It can be used by itself to output the rest of the objects after the skipped ones, or in conjunction with `-First` or `-Last` to return all but the beginning or end of the list of objects. As an example, `-skip` can be used to skip over header lines when reading a file using the `Get-Content` cmdlet.

Limiting the properties of the objects returned

Sometimes, the objects in the pipeline have more properties than you need. To select only certain properties from the list of objects in the pipeline, you can use the `-Property` parameter with a list of properties. For instance, to get only the `Name`, `Extension`, and `Length` from the directory listing, you could do something like this:



The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar button labeled "Script" with a checkmark is visible. The main area contains a command prompt and its output:

```
PS C:\temp> dir C:\temp | select-object -first 5 -Property Name,Length,Extension
```

Name	Length	Extension
PowerShellCookbook		
WORDsearch		
2-ef5-m2-dbfirst-exerci...	1541774	.zip
3-Month Free Trial-6-17...	3775	.publishsettings
adv_event1.ps1	2399	.ps1

Below the table, status information shows "Ln 4450 Col 13" and a zoom level of "100%".

I used the `-First` parameter as well to save some space in the output, but the important thing is that we only got the three properties that we asked for.

Note that, here, the first two objects in the pipeline were directories, and directory objects don't have a length property. PowerShell provides an empty value of `$null` for missing properties like this. Also, note that these objects weren't formatted like a directory listing. We'll talk about formatting in detail in the next chapter, but for now, you should just know that these limited objects are not of the same type as the original objects, so the formatting system treated them differently.

Tip

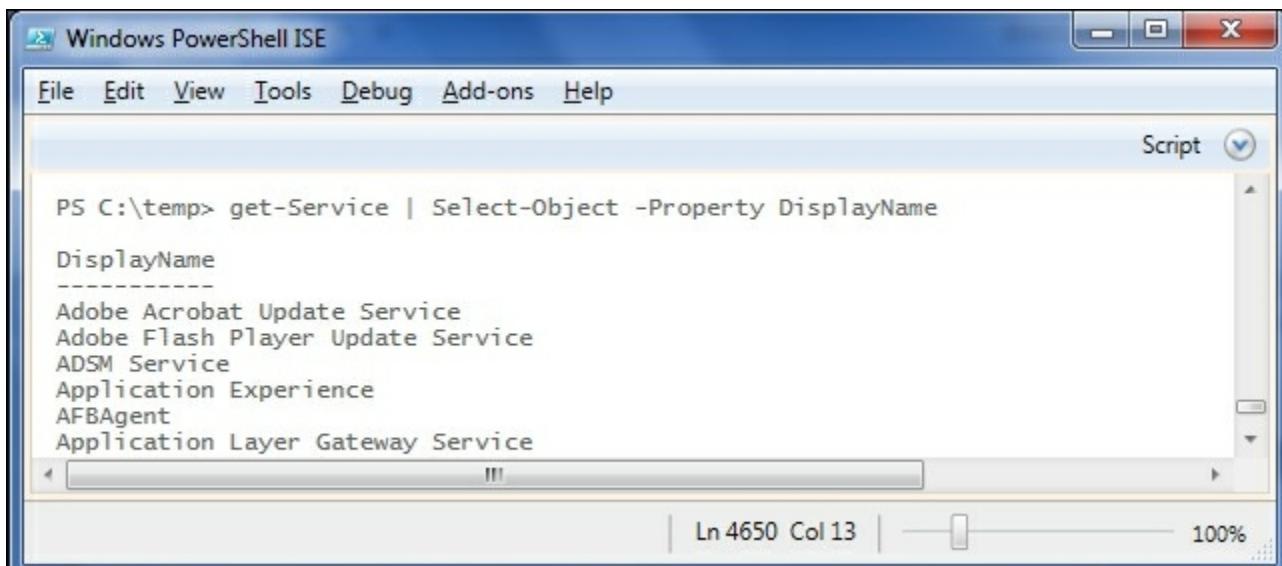
You try it!

Use the `Get-Member` cmdlet to verify that the type of the objects change slightly when you use the `-Property` parameter of `Select-Object`.

Retrieving the values of a single property

Sometimes, you want to get the values of a single property of a set of objects. For instance, if you wanted to get the display names of all of the services installed on your computer, you might try to do something like

this:



Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

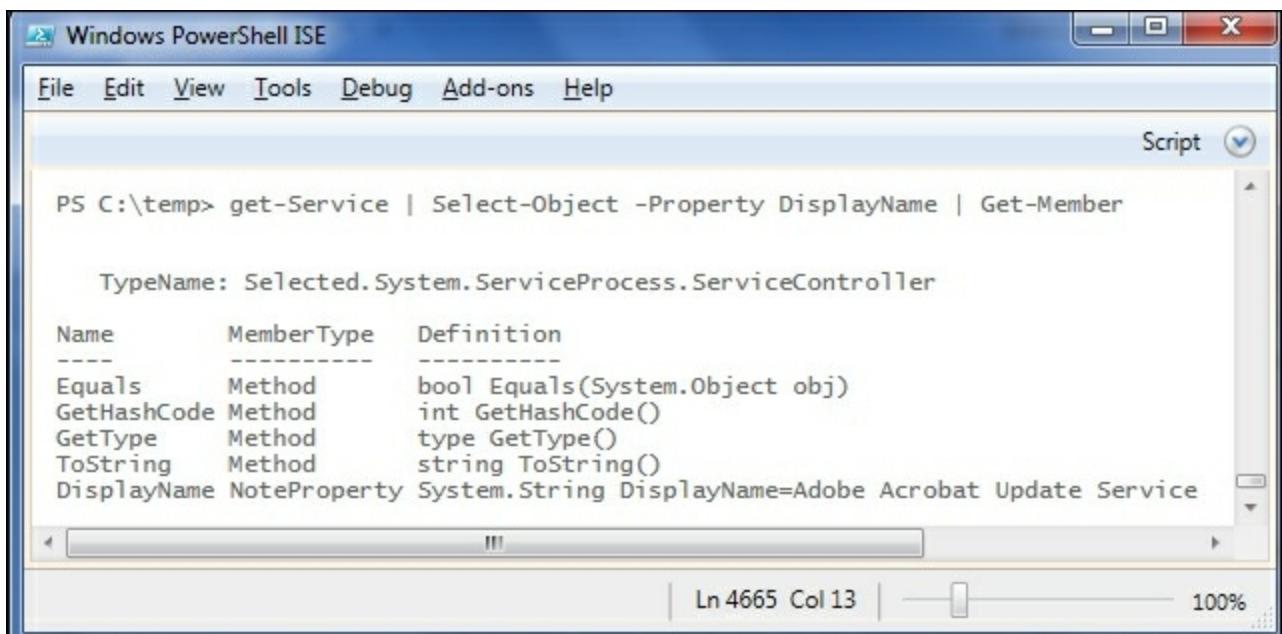
Script

```
PS C:\temp> get-Service | Select-Object -Property DisplayName
```

DisplayName
Adobe Acrobat Update Service
Adobe Flash Player Update Service
ADSM Service
Application Experience
AFBAgent
Application Layer Gateway Service

Ln 4650 Col 13 | 100%

This is close to what you were looking for, but instead of getting a bunch of names (strings), you got a bunch of objects with the `DisplayName` properties. A hint that this is what happened is seen by the heading (and underline) of `DisplayName`. You can also verify this using the `Get-Member` cmdlet:



Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script

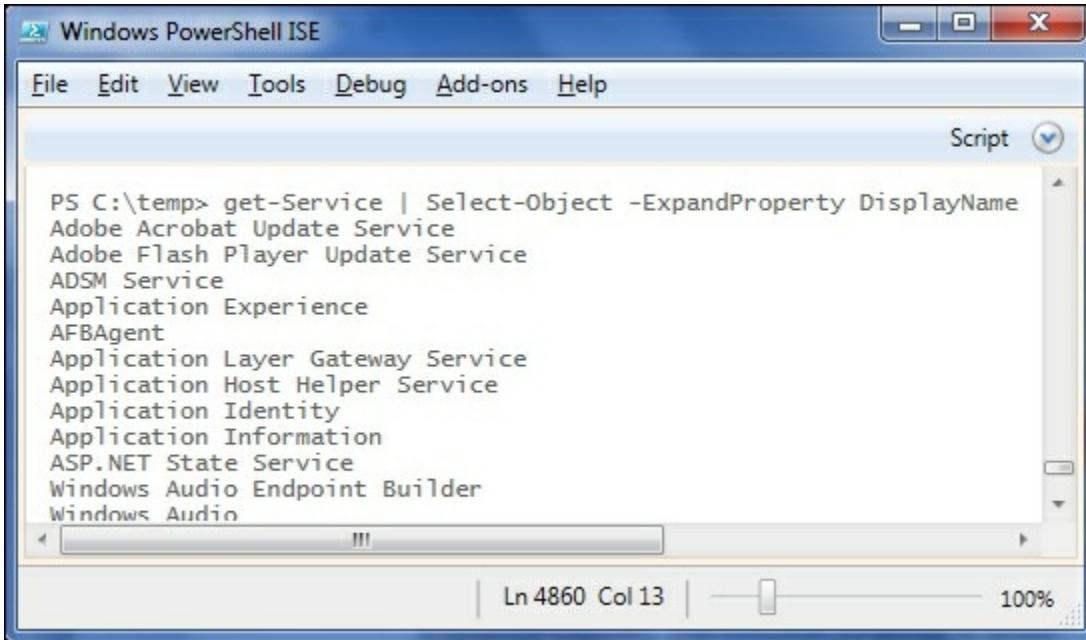
```
PS C:\temp> get-Service | Select-Object -Property DisplayName | Get-Member
```

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
DisplayName	NoteProperty	System.String DisplayName=Adobe Acrobat Update Service

Ln 4665 Col 13 | 100%

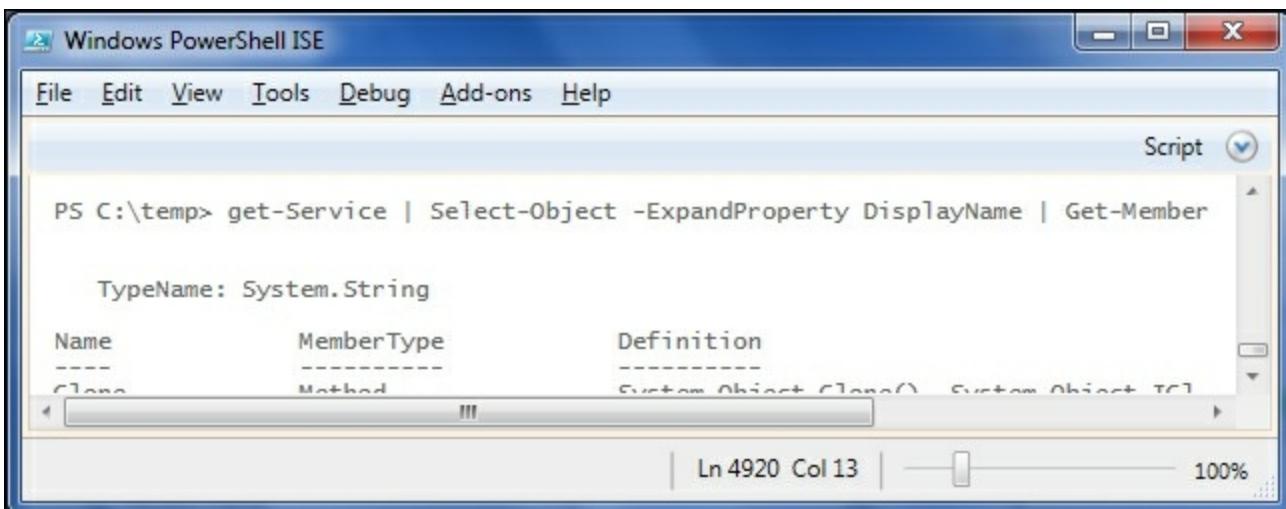
In order to just get the values and not objects, you need to use the `-ExpandProperty` parameter. Unlike the `-Property` parameter, you can

only specify a single property with `-ExpandProperty`, and the output is a list of raw values. Notice that with `-ExpandProperty`, the column heading is gone:



The screenshot shows a Windows PowerShell ISE window. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu labeled "Script" is open. The main pane displays the command "PS C:\temp> get-Service | Select-Object -ExpandProperty DisplayName" followed by a list of service names: Adobe Acrobat Update Service, Adobe Flash Player Update Service, ADSM Service, Application Experience, AFBAgent, Application Layer Gateway Service, Application Host Helper Service, Application Identity, Application Information, ASP.NET State Service, Windows Audio Endpoint Builder, and Windows Audio.

We can also verify using `Get-Member` that we just got strings instead of objects with a `DisplayName` property:



The screenshot shows a Windows PowerShell ISE window. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu labeled "Script" is open. The main pane displays the command "PS C:\temp> get-Service | Select-Object -ExpandProperty DisplayName | Get-Member". Below this, a table titled "TypeName: System.String" is shown with columns "Name", "MemberType", and "Definition". The "Name" column lists "Clone" and "Method". The "MemberType" column lists "Method" and "Method". The "Definition" column lists "System.Object Clone()", "System.Object ToString()", and "System.Object GetHashCode()".

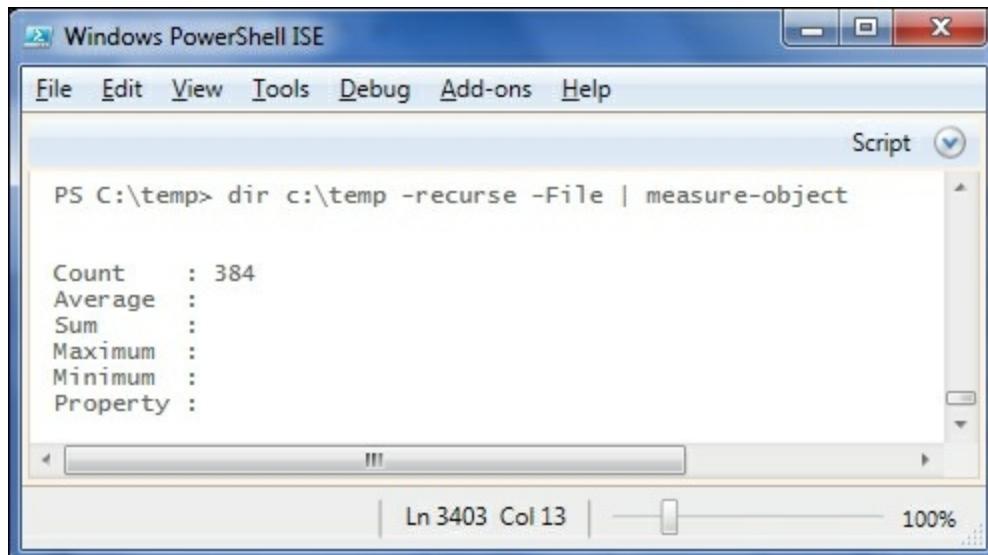
There are a few other parameters for `Select-Object`, but they will be less commonly used than the ones listed here.

The Measure-Object cmdlet

The **Measure-Object** cmdlet has a simple function. It calculates statistics based on the objects in the pipeline. Its most basic form takes no parameters, and simply counts the objects that are in the pipeline.

To count the files in `c:\temp` and its subfolders, you could write:

```
dir c:\temp -recurse -file | Measure-Object
```

A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu shows "Script". The main window displays a command prompt: "PS C:\temp> dir c:\temp -recurse -File | measure-object". Below the command, the output is shown:

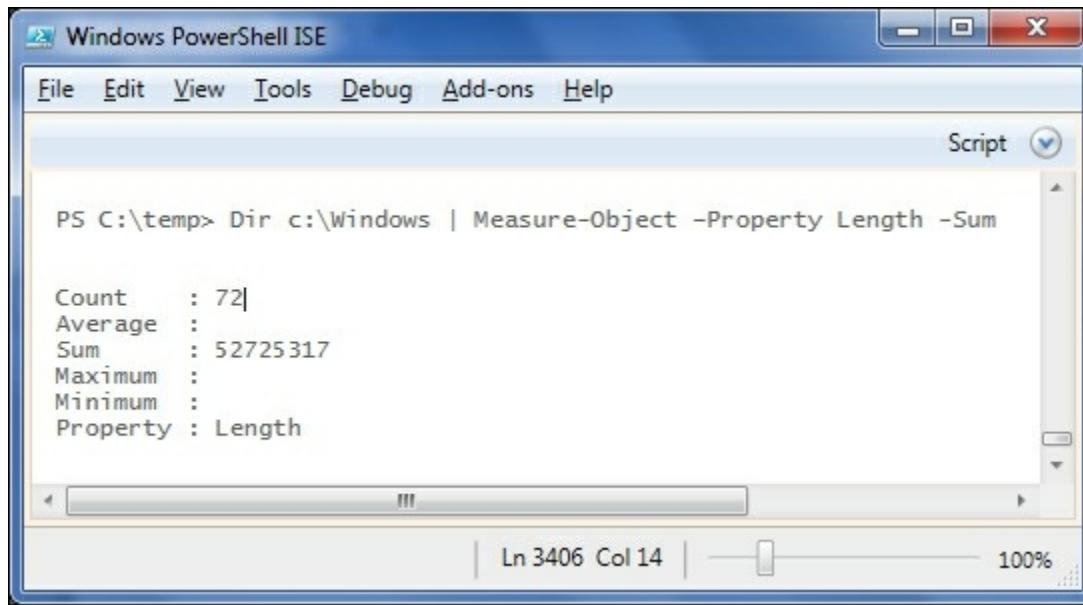
```
Count      : 384
Average    :
Sum        :
Maximum   :
Minimum   :
Property  :
```

At the bottom of the window, there is a status bar with "Ln 3403 Col 13" and a zoom level of "100%".

The output shows the count, and also some other properties that give a hint about the other uses of the cmdlet. To populate these other fields, you will need to provide the name of the property that is used to calculate them and also specify which field(s) you want to calculate. The calculations are specified using the `-Sum`, `-Minimum`, `-Maximum`, and `-Average` switches.

For instance, to add up (sum) the lengths of the files in the `C:\Windows` directory, you could issue this command:

```
Dir c:\Windows | Measure-Object -Property Length -Sum
```



A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The title bar says "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar with a "Script" button is visible. The main window shows a command prompt with the following output:

```
PS C:\temp> Dir c:\Windows | Measure-Object -Property Length -Sum
Count      : 72
Average    :
Sum        : 52725317
Maximum    :
Minimum    :
Property   : Length
```

The status bar at the bottom shows "Ln 3406 Col 14" and a zoom level of "100%".

Tip

You try it!

Use the `Measure-Object` cmdlet to find the size of the largest file on your C:.

The **Group-Object** cmdlet

The **Group-Object** cmdlet divides the objects in the pipeline into distinct sets based on a property or a set of properties. For instance, we can categorize the files in a folder by their extensions using `Group-Object` like this:

```
Administrator: Windows PowerShell
PS C:\temp> dir c:\temp | Group-Object -property Extension
Count Name                           Group
---- 
4 .zip                            {files, PowerShellCookbook, WORDsearch, test}
2 .publishsettings                 {2-eF5-m2-dbFirst-exercise-files.zip, Chapter1_1stDraft.zip}
7 .txt                            {3-Month Free Trial-6-17-2013-credentials.publishsettings}
18 .ps1                           {adv_event1.ps1, datefolder.ps1, get-extension.ps1, get-powershellversionmessage.ps1...}
1 .psm1                          {AreaFunctions.psm1}
3 .html                           {blogs.html, localhost.html, test.html}
2 .opml                          {blogs.opml, blogs2.opml}
2 .xml                           {books.xml, file.xml}
1 .log                            {cbs.log}
1 .ps1xml                        {chapt4_ping_status.ps1xml}
4 .jpg                            {driversLicense.jpg, gwen_birth.jpg, marriage.jpg, release.jpg}
3 .csv                            {FileList.csv, issues.csv, ServerList.csv}
1 .cmd                            {install.cmd}
1 .docx                           {Manuscript Questionnaire 1P - RavenDB in Action.docx}
1 .cer                            {MikeAzure.cer}
5 .xps                            {ricepudding.xps, roses.xps, wallflowers.xps, werthers.xps...}
1 .sql                            {sampledatabase.sql}
1 .clixml                         {SQL кликсмл}
2 .dll                            {System.Management.Automation.dll, System.Management.Automation.Resources.dll}
2 .FDB                            {TESTADO (2).FDB, TESTADO.FDB}
1 .psd1                           {testmanifest.psd1}
1 .exe                            {visualStudioHelpDownloader2012.exe}
1 .config                         {visualStudioHelpDownloader2012.exe.config}
1 .chm                            {WindowsPowerShell12.0CoreHelp-May2011.chm}

PS C:\temp>
```

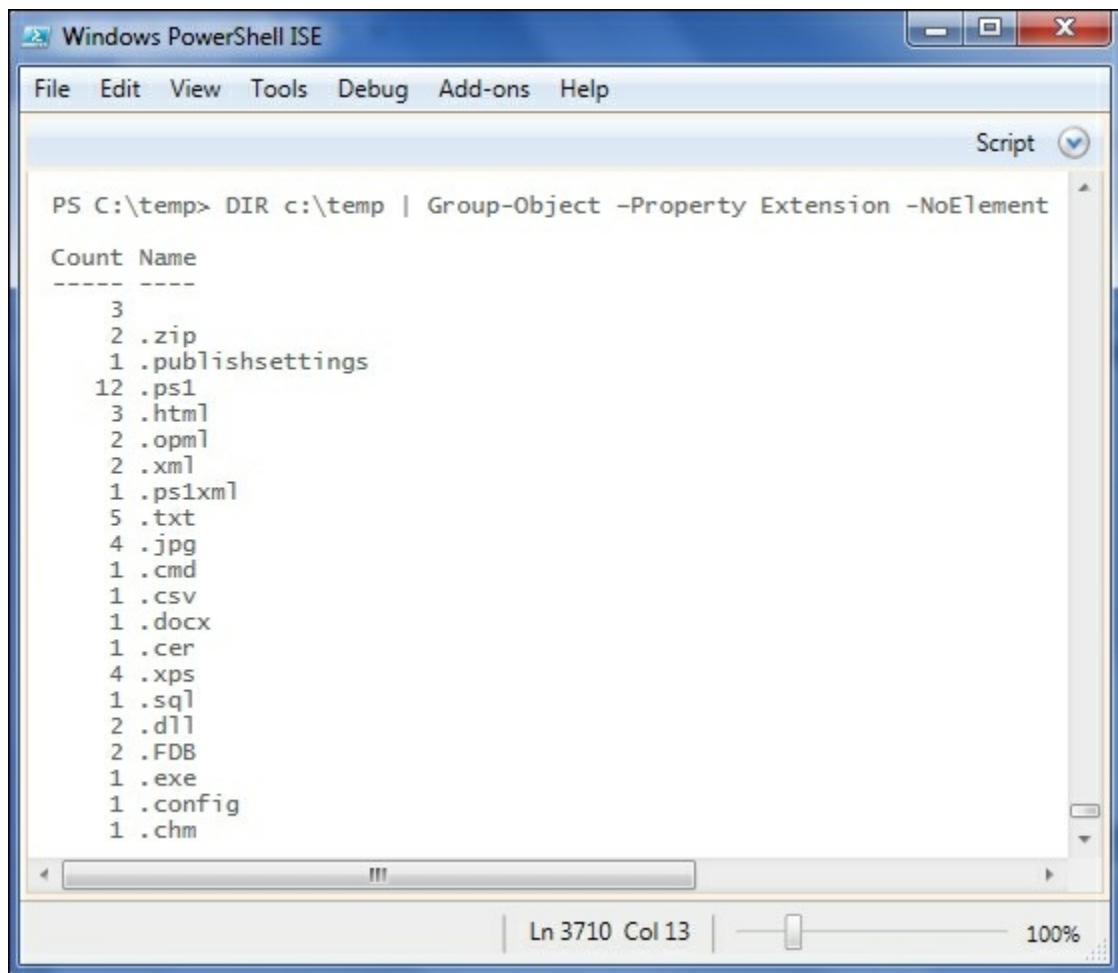
You will notice in the output that PowerShell has provided the count of items in each set, the value of the property (Extension) that labels the set, and a property called **Group**, which contains all of the original objects that were on the pipeline that ended up in the set. If you have used the `GROUP BY` clause in SQL and are used to losing the original information when you group, you'll be pleased to know that PowerShell retains those objects in their original state in the `Group` property of the output.

Tip

You try it!

Can you think of a way to get the original objects out of the `Group` property? (Hint: one of the ways to use `Select-Object` might come in handy here.)

If you don't need the objects and are simply concerned with what the counts are, you can use the `-NoElement` switch, which causes the `Group` property to be omitted.



A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The title bar reads "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar button labeled "Script" with a dropdown arrow is visible. The main window displays a command-line output:

```
PS C:\temp> DIR c:\temp | Group-Object -Property Extension -NoElement
```

Count	Name
3	.zip
1	.publishsettings
12	.ps1
3	.html
2	.opml
2	.xml
1	.ps1xml
5	.txt
4	.jpg
1	.cmd
1	.csv
1	.docx
1	.cer
4	.xps
1	.sql
2	.dll
2	.FDB
1	.exe
1	.config
1	.chm

The status bar at the bottom shows "Ln 3710 Col 13" and "100%".

You're not limited to grouping by a single property either. If you want to see files grouped by mode (read-only, archive, etc.) and extension, you can simply list both properties.

The screenshot shows a Windows PowerShell ISE window titled "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar above the main area has a "Script" button. The main pane displays the following PowerShell command and its output:

```
PS C:\temp> dir | group-object Mode,Extension -NoElement
```

Count	Name
2	d----
2	-a---, .zip
1	-a---, .publishsettings
12	-a---, .ps1
3	-a---, .html
2	-a---, .opml
2	-a---, .xml
1	-a---, .ps1xml
4	-a---, .txt
4	-a---, .jpg
1	-----, .cmd
1	-a---, .csv
1	-a---, .docx
1	-a---, .cer
1	-ar--, .xps
3	-a---, .xps
1	-a---, .sql
2	-a---, .dll
1	-a---,
2	-a---, .FDB
1	-a---, .exe
1	-a---, .config
1	-a---, .chm

The status bar at the bottom shows "Ln 4387 Col 13" and a zoom level of "100%".

Note that the `Name` property of each set is now a list of two values corresponding to the two properties defining the group.

The `Group-Object` cmdlet can be useful to summarize the objects, but you will probably not use it nearly as much as `Sort-Object`, `Where-Object`, and `Select-Object`.

Putting them together

The several `*-Object` cmdlets that we've discussed in this chapter will be the foundation to your experience with PowerShell. Although, we only used them with `Dir`, `Get-Service`, and `Get-Process`, this is only because I can be sure that you can use these cmdlets on any system. The way that you use the `*-Object` cmdlets is the same whether you're dealing with files and folders, virtual machines, or mailboxes in Exchange. Since PowerShell gives you objects in all of these situations, these cmdlets will enable you to manipulate them using the same techniques.

Here are a couple of concrete examples of how the methods of using these cmdlets are portable between the types of objects. First, to get the largest five files in the `c:\Windows` folder, you would do this:

```
Dir c:\Windows | Sort-Object -Property Length -Descending |  
Select-Object -First 5
```

Similarly, getting the five processes that are using the most file handles would look like this:

```
Get-Process | Sort-Object -Property Handles -Descending |  
Select-Object -First 5
```

Can you see that the methodology to solve these two problems is exactly the same? Can you think of other problems that you might solve using a similar approach?

Tip

You Try it!

Use the cmdlets in this chapter to find the five newest files on your `c:` drive (use the `LastWriteTime` property to determine the age).

Summary

In this chapter, we looked at how to use the pipeline in PowerShell and several of the cmdlets that deal with objects in general. Once you learn to use these cmdlets efficiently, you will be able to solve all kinds of interesting problems in many areas.

In the next chapter, we will investigate PowerShell's versatile formatting system and learn how to format output in different ways.

For further reading

- Get-Help about_pipelines
- Get-Help Sort-Object
- Get-Help Where-Object
- Get-Help Select-Object
- Get-Help Group-Object
- Get-Help about_comparison_operators
- Get-Help about_scriptblocks

Chapter 5. Formatting Output

So far, we are aware of PowerShell formatting the output of the commands that we execute, but we haven't spent any time on the particular ways that the output is formatted. That changes in this chapter. Now, we will learn all about the PowerShell formatting system and how we can take advantage of it. The topics covered in this chapter include the following:

- When does formatting occur?
- The rules of automatic formatting
- The cmdlets that control formatting
- The dangers of formatting
- The best practices of formatting

When does formatting occur?

The first thing to understand about PowerShell formatting is that the host you are in always formats the output. Remember that the host is the application (for example, the PowerShell console or the ISE) that is running the PowerShell engine and executing the commands that you enter. Since we know that cmdlets always output objects, the presence of some kind of formatting system is clear when we look at the output of a cmdlet such as `dir` (or `Get-ChildItem`):

The screenshot shows a Windows PowerShell ISE window. The title bar reads "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar button labeled "Script" with a dropdown arrow is visible. The main area displays the command "PS C:\Users\Mike> dir c:\". Below this, the output shows the contents of the C:\ directory in a tabular format:

Mode	LastWriteTime	Length	Name
d----	8/19/2013 9:39 PM		develop
d----	7/9/2012 8:07 PM		develop-archive
d----	4/2/2015 8:12 PM		develop-powershell
d----	1/16/2011 3:51 PM		develop-python
d----	11/15/2010 4:26 AM		eSupport
d----	3/14/2015 4:01 PM		GOG.com
d----	12/22/2010 9:26 PM		inetpub

The status bar at the bottom indicates "Ln 34 Col 19" and "100%".

Somehow, certain properties are displayed and others are hidden. In this output, we can even see that the objects are grouped by a particular property (`PSParentPath`) by the display of **Directory: C:** at the top.

When the host executes a pipeline (and any line of code is a pipeline), it appends the `Out-Default` cmdlet to the end of the pipeline. The `Out-Default` cmdlet's job is to send the objects to the default formatter. The default formatter follows a set of rules to convert the output objects into special formatting objects. The host knows how to output the formatting objects, so you can see the formatted data. The trick to formatting in PowerShell is to know the rules.

Tip

If you go down the advanced route and write your own host application, you have a control over whether `Out-Default` is called. For instance, if you wanted to display output in a grid rather than in text, you wouldn't want the formatting system to get in the way and format the objects for you.

The rules of automatic formatting

PowerShell formatting is somewhat complex and can be confusing at first. In fact, there are only a few basic rules:

- Format files specify the formatting for specific types
- Formatting decisions are based on the first object in the pipeline
- Objects with four or less properties are formatted as tables
- Objects with five or more properties are formatted as lists

Formatting files

In `$PSHOME`, there are files whose names end with `format.ps1xml`. These files are XML files that specify the default formatting specifications for many of the types of objects that you will work with in PowerShell.

These files are loaded by the host when the PowerShell session starts. If you write your own format files, you can load them using the `Update-FormatData` cmdlet. The details of writing your own format files are beyond the scope of Module 1, but a quick read through the `FileSystem.Format.ps1xml` file will make some things clear:

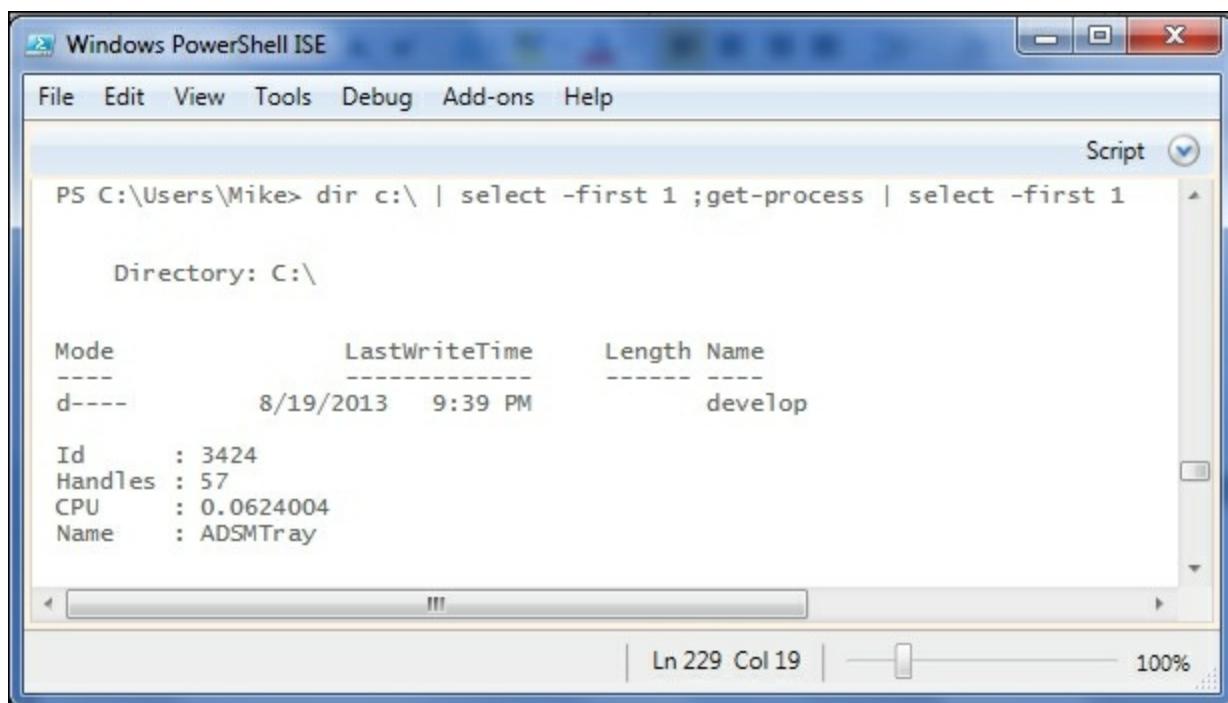
```
FileSystem.format.ps1xml X
17 <Configuration>
18   <SelectionSets>
19     <SelectionSet>
20       <Name>FileSystemTypes</Name>
21       <Types>
22         <TypeName>System.IO.DirectoryInfo</TypeName>
23         <TypeName>System.IO.FileInfo</TypeName>
24       </Types>
25     </SelectionSet>
26   </SelectionSets>
27
28   <!-- ##### GLOBAL CONTROL DEFINITIONS ##### -->
29   <Controls>
30     <Control>
31       <Name>FileSystemTypes-GroupingFormat</Name>
32       <CustomControl>
33         <CustomEntries>
34           <CustomEntry>
35             <CustomItem>
36               <Frame>
```

In this file, first, we can see that the formatting is selected by the objects of the `System.IO.DirectoryInfo` and `System.IO.FileInfo` types.

Second, the first view specified is grouped by `PSParentPath` and we can see the properties listed (`Mode`, `LastWriteTime`, `Length`, and `Name`) in the table format. These two observations match what we see, when we execute the `Get-ChildItem` cmdlet against a file system path.

Formatting decisions are based on the first object

Knowing that the type of objects are matched against the formatting specified in the files is part of the equation. The next thing to know is that PowerShell relies on the first object in the pipeline to determine what formatting to apply. If subsequent objects don't "fit" in that format, PowerShell will revert to a list display, listing the property names and values separated by a colon. To illustrate this, we can join two different pipelines together using a semicolon. This will cause the output of both pipelines to be treated as if they were from the same pipeline. The first part of the pipeline is outputting the first item in the root of `C:` and the second is outputting the first process:



The screenshot shows a Windows PowerShell ISE window. The command entered is:

```
PS C:\Users\Mike> dir c:\ | select -first 1 ;get-process | select -first 1
```

The output is:

```
Directory: C:\

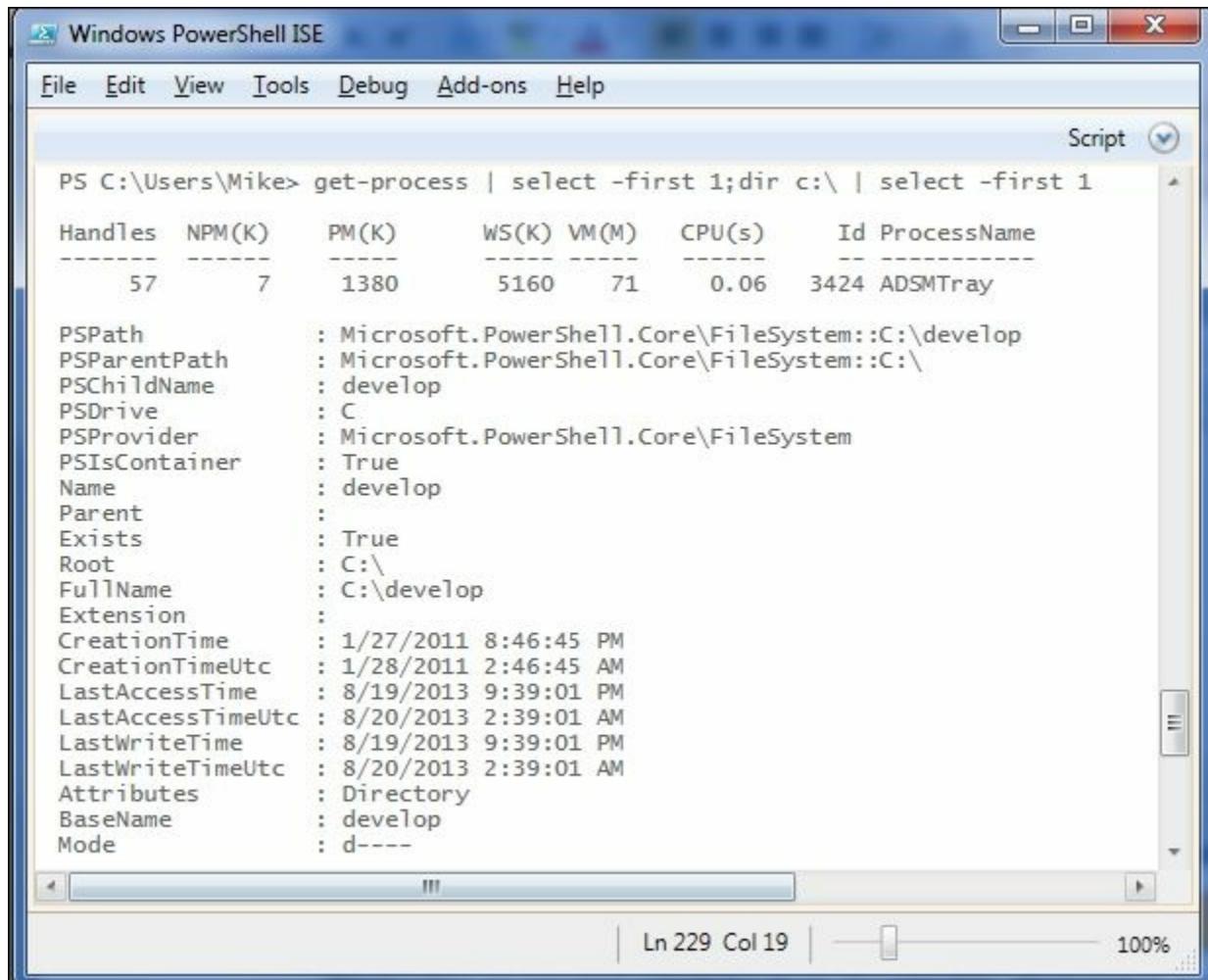
Mode          LastWriteTime      Length Name
----          -----          ---- 
d---  8/19/2013 9:39 PM          develop

Id : 3424
Handles : 57
CPU : 0.0624004
Name : ADSMTray
```

The pipeline we executed outputted a single `DirectoryInfo` object as well as a `process` object. PowerShell selected the default, tabular view

for the `DirectoryInfo` object, but the `Process` object doesn't fit in this format. Thus, the `Process` object was formatted in a generic way, listing the properties one per line.

If we reverse the pipeline, we'll see that the `Process` object gets its specified format and the `DirectoryInfo` object is generically treated:



The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A dropdown menu is open under "File" with "Script" selected. The main area displays the following PowerShell command and its output:

```
PS C:\Users\Mike> get-process | select -first 1;dir c:\ | select -first 1
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
57	7	1380	5160	71	0.06	3424	ADSMTray

```
PSPPath          : Microsoft.PowerShell.Core\FileSystem::C:\develop
PSParentPath     : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName      : develop
PSDrive          : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer    : True
Name             : develop
Parent           :
Exists           : True
Root             : C:\
FullName         : C:\develop
Extension        :
CreationTime     : 1/27/2011 8:46:45 PM
CreationTimeUtc  : 1/28/2011 2:46:45 AM
LastAccessTime   : 8/19/2013 9:39:01 PM
LastAccessTimeUtc: 8/20/2013 2:39:01 AM
LastWriteTime    : 8/19/2013 9:39:01 PM
LastWriteTimeUtc: 8/20/2013 2:39:01 AM
Attributes       : Directory
BaseName         : develop
Mode             : d----
```

The status bar at the bottom shows "Ln 229 Col 19" and "100%".

Another thing that can be confusing is that if we use `Select-Object` to limit the properties output, the resulting object is no longer of the original type. For example, we execute the following:

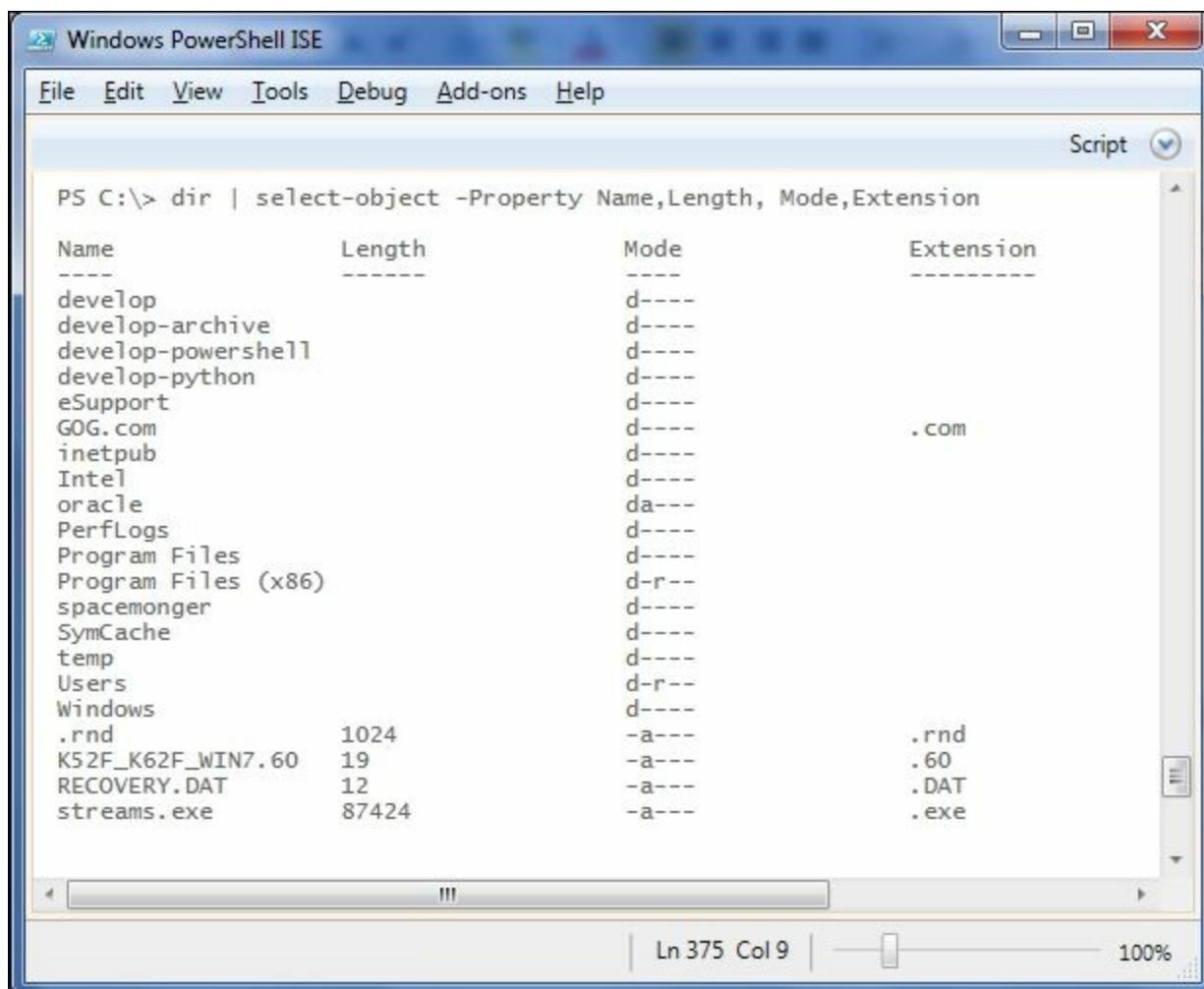
```
Dir | Select-Object -Property Name,Length
```

Here, we would no longer be outputting the `System.IO.FileInfo` and `System.IO.DirectoryInfo` objects, but the `Selected.System.IO.FileInfo` and `Selected.System.IO.DirectoryInfo`

objects. These new objects would not be selected by the same format files, because the type is different.

Small objects go in a table

If the objects being formatted don't match a format file, one of the rules of formatting says that small objects with four or less properties are formatted as a table. We can verify this using `Select-Object` as mentioned in the last section to create new objects that don't match the existing format file:



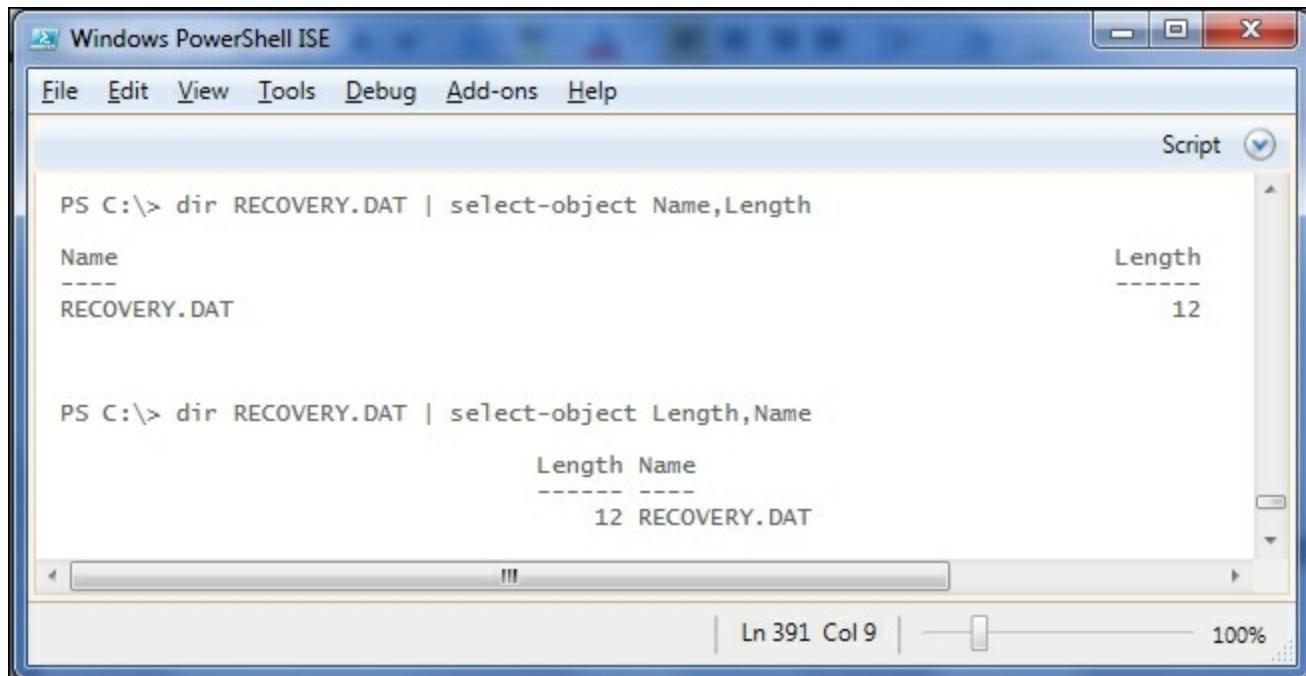
The screenshot shows a Windows PowerShell ISE window. The title bar reads "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu is open under the "Script" button. The main area displays a command prompt and its output. The command is "PS C:\> dir | select-object -Property Name,Length,Mode,Extension". The output is a table with four columns: Name, Length, Mode, and Extension. The table lists various directory entries like "develop", "develop-archive", etc., along with their file sizes and attributes. The "Extension" column shows ".com" for GOG.com and ".rnd" for .rnd.

Name	Length	Mode	Extension
---	-----	-----	-----
develop		d----	
develop-archive		d----	
develop-powershell		d----	
develop-python		d----	
eSupport		d----	
GOG.com		d----	.com
inetpub		d----	
Intel		d----	
oracle		da---	
PerfLogs		d----	
Program Files		d----	
Program Files (x86)		d-r--	
spacemonger		d----	
SymCache		d----	
temp		d----	
Users		d-r--	
Windows		d----	
.rnd	1024	-a---	.rnd
K52F_K62F_WIN7.60	19	-a---	.60
RECOVERY.DAT	12	-a---	.DAT
streams.exe	87424	-a---	.exe

As expected, not only do we avoid the grouped format that we saw before for `Dir` output, but we also got the data in a table since there were only four properties.

Sometimes, we find that the table format that is generated doesn't look

very nice. The engine divides the line evenly into the number of columns that are required. It also formats each column according to the type of data in the properties of the first object, left-aligning strings and right-aligning numbers. Look at the output of the two commands in the following screenshot, and you will see that the spacing of both of the tables is looking somewhat ridiculous. We'll see later in the chapter what we can do to improve this situation, but for now, we just need to be aware that sometimes default formatting is less than perfect:



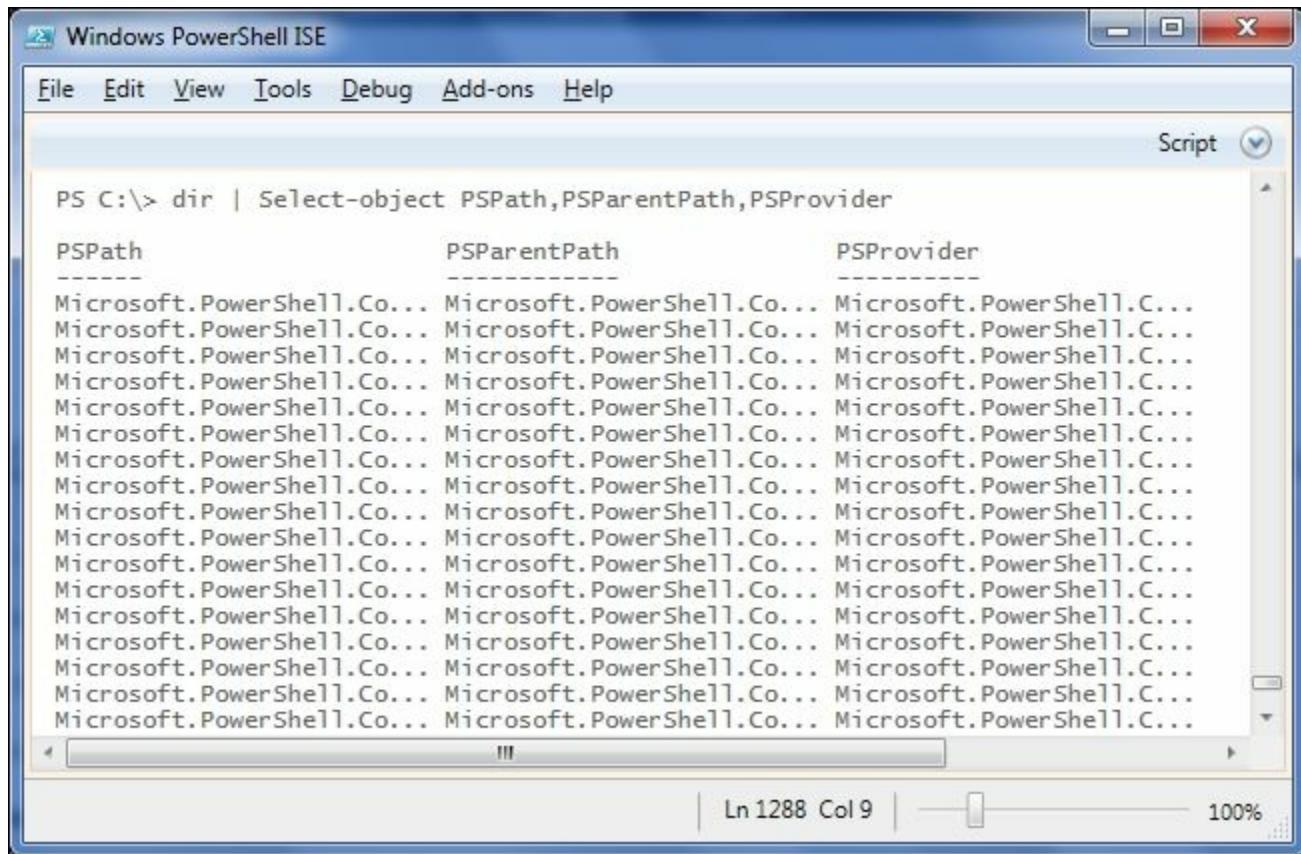
The screenshot shows a Windows PowerShell ISE window with a blue title bar and a menu bar with File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with a 'Script' button is visible. The main area contains two command-line outputs:

```
PS C:\> dir RECOVERY.DAT | select-object Name,Length
Name
-----
RECOVERY.DAT
Length
-----
12

PS C:\> dir RECOVERY.DAT | select-object Length,Name
Length Name
-----
12 RECOVERY.DAT
```

At the bottom, status information shows Ln 391 Col 9 and a zoom level of 100%.

Another issue with formatting in a table is that sometimes the properties don't fit very well in the columns. When this happens, PowerShell will truncate the value and end the display with an ellipsis (...):



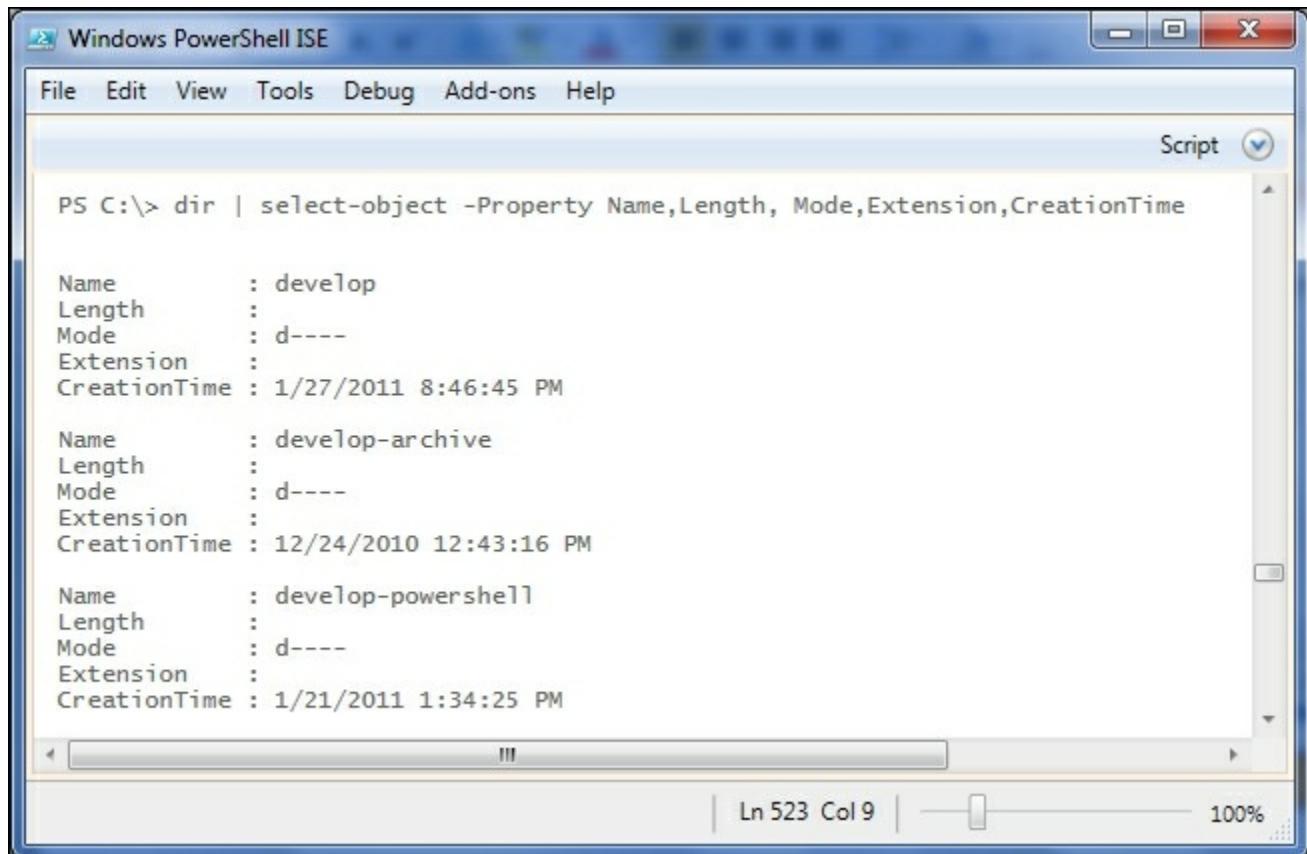
A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The title bar reads "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar with a "Script" button and a dropdown arrow is visible. The main window displays a command prompt: "PS C:\> dir | Select-object PSPath,PSParentPath,PSProvider". Below the command, a table is shown with three columns: "PSPATH", "PSProviderPath", and "PSProvider". The "PSPATH" column contains 25 entries all starting with "Microsoft.PowerShell.Co...". The "PSProviderPath" and "PSProvider" columns also contain 25 entries, all starting with "Microsoft.PowerShell.C...". The bottom status bar shows "Ln 1288 Col 9" and a zoom level of "100%".

PSPATH	PSProviderPath	PSProvider
Microsoft.PowerShell.Co...	Microsoft.PowerShell.Co...	Microsoft.PowerShell.C...

Large objects go in a list

Objects that are larger than four properties are placed into what PowerShell calls a list format, placing each property of each object in a separate line, and separating the property name from the property value with a colon. We saw this format earlier when an object did not fit in the format selected by the first object in the pipeline.

Continuing the examples using `Dir`, we can see this format in action. Adding a fifth property to the `Select-Object` cmdlet causes the output to be formatted as a list:



A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with a "Script" button and a dropdown arrow is visible. The main window displays a command prompt: PS C:\> dir | select-object -Property Name,Length, Mode,Extension,CreationTime. The output is a list of file properties:

```
Name      : develop
Length    :
Mode      : d-----
Extension :
CreationTime : 1/27/2011 8:46:45 PM

Name      : develop-archive
Length    :
Mode      : d-----
Extension :
CreationTime : 12/24/2010 12:43:16 PM

Name      : develop-powershell
Length    :
Mode      : d-----
Extension :
CreationTime : 1/21/2011 1:34:25 PM
```

The status bar at the bottom shows "Ln 523 Col 9" and a zoom level of "100%".

List formats can seem cleaner than tabular formats because there is less room for issues with column widths and left and right justification. On the other hand, the output is spread over a lot more lines and is harder to scan visually.

Tip

Since default formats can cause properties to be omitted from the output, it is often convenient to use `Select-Object` to create new "Selected" objects that skip these formats. To see all the properties, you can use a wildcard (such as `*`) to match everything.

Try this, for example: `Dir | Select-Object -Property *`

Cmdlets that control formatting

The rules in the previous section covered all the output where we haven't specifically told PowerShell how to format. What is nice here is that the default formats for many of the types that are commonly encountered in PowerShell do a good job of displaying the most often used properties, and the rules to format other objects are also, generally, very appropriate. But in some circumstances, we want to have more control over the output, and it's no surprise that PowerShell provides cmdlets for this purpose.

Format-Table and Format-List

The two most commonly used formatting cmdlets are `Format-Table` and `Format-List`. As their names suggest, they force the formatting system to use either a table or a list format for the objects, irrespective of the number of properties or the type-related formatting files that might be present. Each of the cmdlets has a `-Property` parameter that takes a list of properties to be included. A couple of quick examples should make the basic usage clear.

First, we can use `Format-List` with the long property value example we saw earlier, which led to an unusable table, to see how this would look like as a list:

```
Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script

PS C:\> dir | Format-List -property PSPath,PSParentPath,PSProvider

PSPath      : Microsoft.PowerShell.Core\FileSystem::C:\develop
PSProvider  : Microsoft.PowerShell.Core\FileSystem
PSProvider  : Microsoft.PowerShell.Core\FileSystem

PSPath      : Microsoft.PowerShell.Core\FileSystem::C:\develop-archive
PSProvider  : Microsoft.PowerShell.Core\FileSystem
PSProvider  : Microsoft.PowerShell.Core\FileSystem

PSPath      : Microsoft.PowerShell.Core\FileSystem::C:\develop-powershell
PSProvider  : Microsoft.PowerShell.Core\FileSystem
PSProvider  : Microsoft.PowerShell.Core\FileSystem

Ln 1378 Col 9 | 100%
```

The screenshot shows the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar with a "Script" button is visible. The main window displays the command "PS C:\> dir | Format-List -property PSPath,PSParentPath,PSProvider" and its output. The output lists three directory objects with their paths and providers. The status bar at the bottom shows "Ln 1378 Col 9" and "100%".

Clearly, the output makes a lot more sense as a list.

Also, we can use `Format-Table` to output a series of objects that would, normally, have been formatted as a list because they have more than four properties. Since we know the properties in this case are numeric, formatting as a table is reasonable:

```
Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script

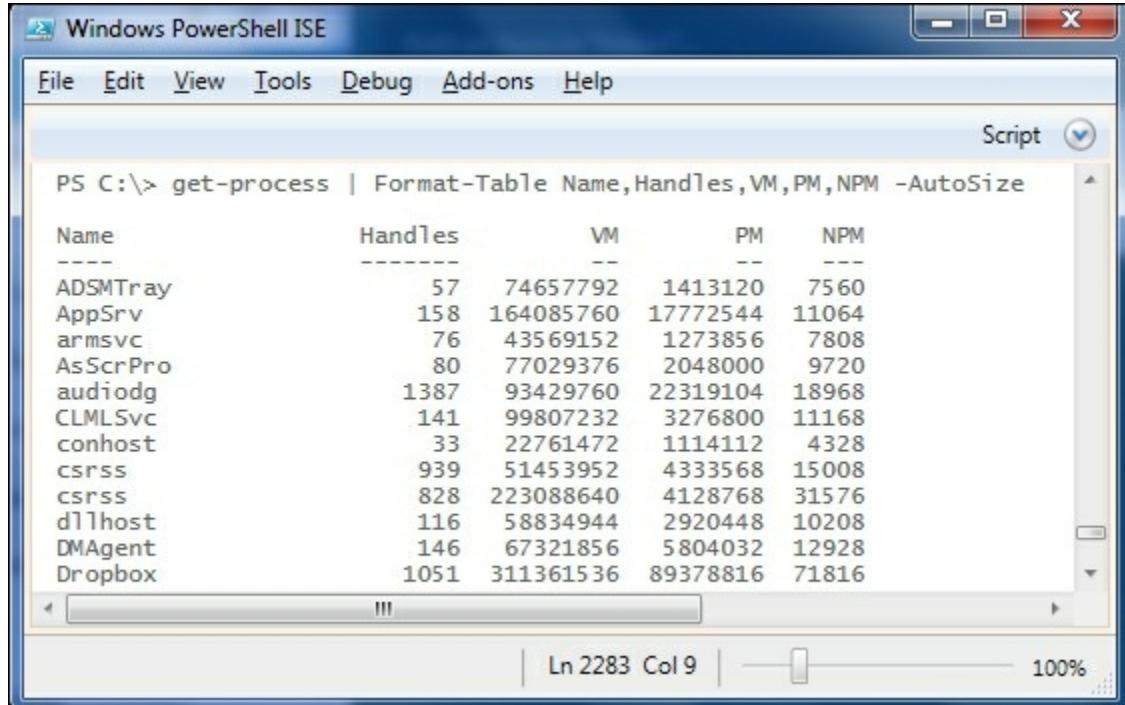
PS C:\> get-process | Format-Table Name,Handles,VM,PM,NPM

Name          Handles    VM        PM        NPM
---          -----
ADSMTray      57        74657792  1413120   7560
AppSrv        158       164085760  17772544  11064
armsvc        76         43569152  1273856   7808
AsscrPro      80         77029376  2048000   9720
audiodg       1387      93429760  22319104  18968
CLMLSvc       141       99807232  3276800   11168
conhost       33         22761472  1114112   4328
csrss         934       51453952  4333568   15008
csrss         829       223109120 4132864   31576
dllhost       116       58834944  2920448   10208
DMAgent       146       67321856  5804032   12928

Ln 2175 Col 9 | 100%
```

The screenshot shows the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar with a "Script" button is visible. The main window displays the command "PS C:\> get-process | Format-Table Name,Handles,VM,PM,NPM" and its output. The output is presented as a table with columns for Name, Handles, VM, PM, and NPM. The status bar at the bottom shows "Ln 2175 Col 9" and "100%".

In this case, we still have some open spaces, so we can utilize the `-AutoSize` switch parameter to improve the output format even more. `AutoSize` tells the formatting system to look at all the values in all the columns before deciding on the column sizes (instead of dividing the line evenly):



The screenshot shows a Windows PowerShell ISE window with the title bar "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu is open under the "Script" button. The command entered in the console is "PS C:\> get-process | Format-Table Name,Handles,VM,PM,NPM -AutoSize". The resulting table has five columns: Name, Handles, VM, PM, and NPM. The table lists various system processes with their corresponding resource usage values.

Name	Handles	VM	PM	NPM
ADSMTray	57	74657792	1413120	7560
AppSrv	158	164085760	17772544	11064
armsvc	76	43569152	1273856	7808
AsScrPro	80	77029376	2048000	9720
audiodg	1387	93429760	22319104	18968
CLMLSvc	141	99807232	3276800	11168
conhost	33	22761472	1114112	4328
cssrss	939	51453952	4333568	15008
cssrss	828	223088640	4128768	31576
dllhost	116	58834944	2920448	10208
DMAgent	146	67321856	5804032	12928
Dropbox	1051	311361536	89378816	71816

If the number of objects is not very high, using `AutoSize` can make the tables much nicer to look at. The `-AutoSize` switch will also help the awkward looking tables, which we saw with two columns, either at the margins or crowded at the center of the page:

Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script

```
PS C:\> dir RECOVERY.DAT | Format-Table Name,Length -AutoSize
Name          Length
----          -----
RECOVERY.DAT      12

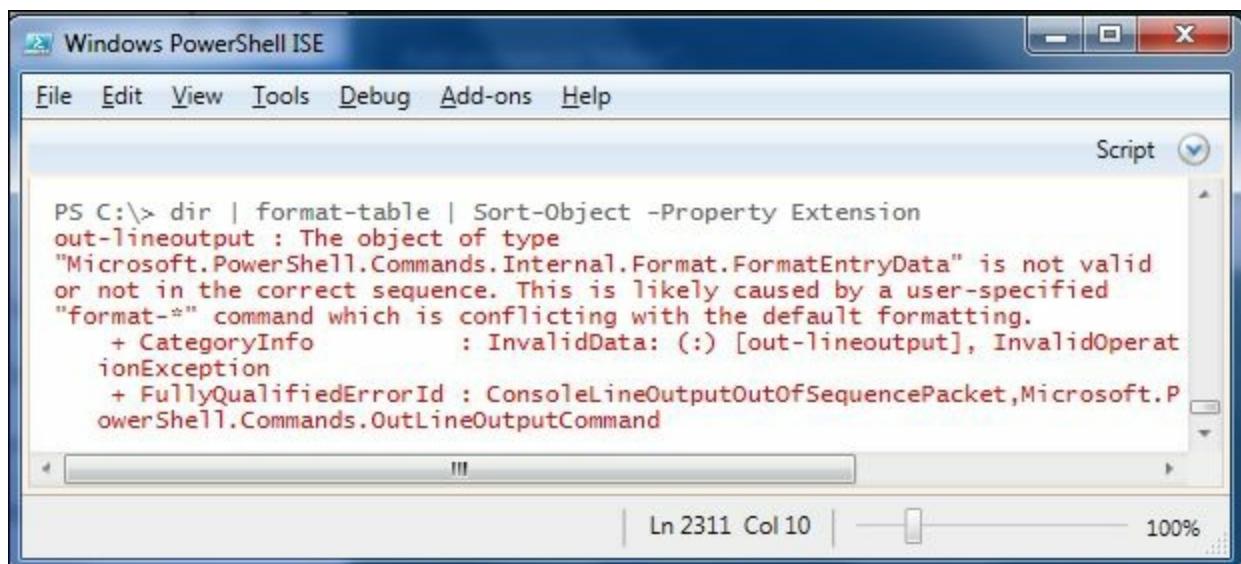
PS C:\> dir RECOVERY.DAT | Format-Table Length,Name -AutoSize
Length Name
----- -----
 12 RECOVERY.DAT
```

Ln 2299 Col 9 | 100%

The dangers of formatting

The automatic formatting of objects makes the PowerShell experience much more comfortable. Seeing familiar looking output and commonly used properties is a great aid to productivity. However, formatting comes with a price. Simply put, formatted objects are not able to be used like the original objects were. For instance, once objects are formatted, there is no way to sort or filter them based on properties. Formatting can be compared with taking pictures of the objects. The picture looks similar to the objects, but are not the actual objects. The only thing that can be done with the formatted objects is to output them somewhere. As mentioned earlier, the `Out-Default` cmdlet sends them to the console. Additionally, the `Out-File`, `Out-Printer`, and `Out-String` cmdlets send the output (obviously) to a file, printer, and string.

Attempting to use the formatted objects in the same way that we've grown accustomed to using objects, fails miserably, as seen from the following screenshot:



A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The window title is "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A dropdown menu labeled "Script" is open. The main pane displays a command prompt and its output. The command entered is `PS C:\> dir | format-table | Sort-Object -Property Extension`. The output shows an error message in red text:
out-lineoutput : The object of type
"Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData" is not valid
or not in the correct sequence. This is likely caused by a user-specified
"format-*" command which is conflicting with the default formatting.
+ CategoryInfo : InvalidData: (:) [out-lineoutput], InvalidOperationException
+ FullyQualifiedErrorId : ConsoleLineOutputOutOfSequencePacket,Microsoft.P
owerShell.Commands.OutLineOutputCommand

Best practices of formatting

Since formatting objects leads to objects that can no longer be manipulated, it is a best practice to avoid formatting anything that you want to be able to manipulate. This seems obvious, but it is easy to get wrong. We will talk about this best practice at length in the chapters covering Scripts and Functions.

Summary

In this chapter, we have walked through the PowerShell formatting system. We saw that PowerShell hosts use `Out-Default` to cause formatting to happen to all the output. We looked at the rules to format objects, when no explicit formatting instructions are included, as well as the cmdlets to tell PowerShell exactly how we want objects to be formatted. Finally, we discovered that formatting is something of a destructive operation, in which the original objects are no longer available after the formatting has occurred.

In the next chapter, we will take our first steps towards packaging sequences of commands for reuse in scripts.

For further reading

- Get-Help about_format.ps1xml
- Get-Help Out-Default
- Get-Help Format-Table
- Get-Help Format-List
- Get-Help Update-FormatData

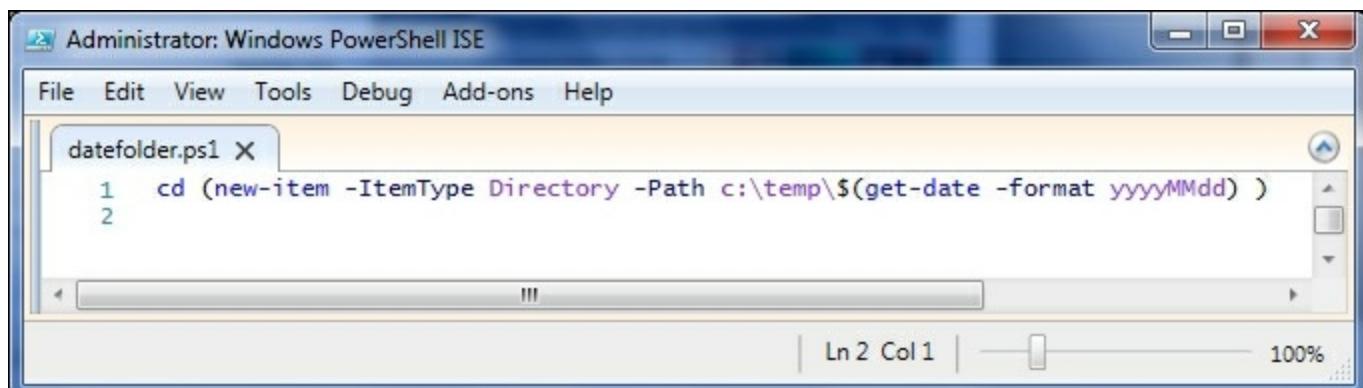
Chapter 6. Scripts

Now that we have learned some useful commands, it would be nice to be able to record a sequence of commands, so that we can execute them all at once. In this chapter, we will learn how to accomplish this with scripts. We will also cover the following topics:

- Execution policies
- Adding parameters to scripts
- Control structures
- Profiles

Packaging commands

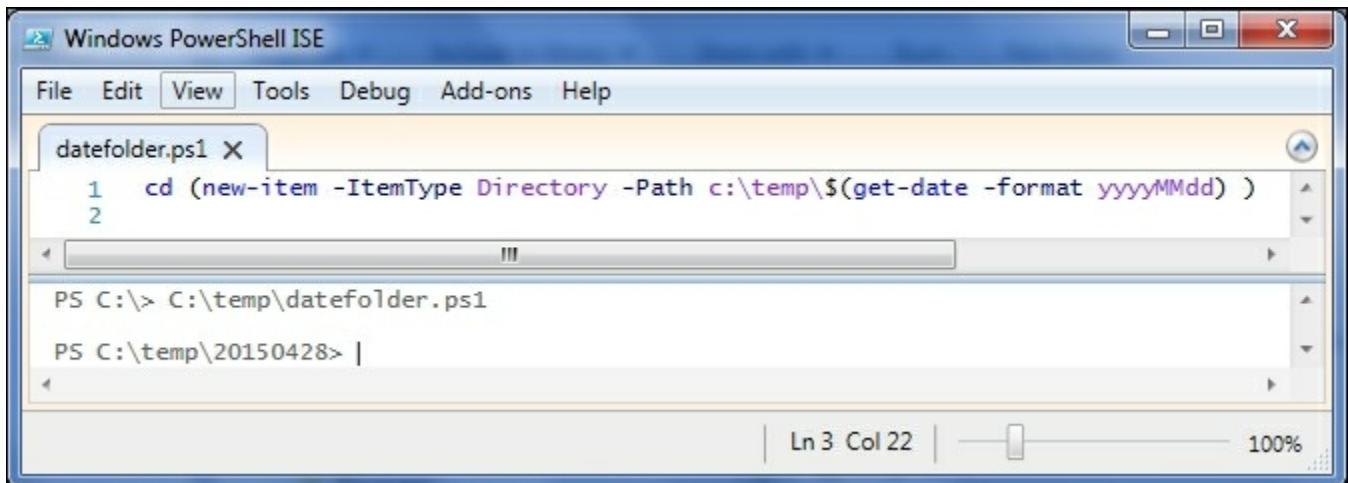
Saving commands in a file for reuse is a pretty simple idea. In PowerShell, the simplest kind of these files is called a script and it uses the `.ps1` extension, no matter what version of PowerShell you're using. For example, if you wanted to create a new folder, under `c:\temp`, with the current date as the name and then change to that folder, you could put these commands in a file:



The screenshot shows the Windows PowerShell Integrated Scripting Environment (ISE) window. The title bar reads "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The main area displays a script named "datefolder.ps1" with the following content:

```
cd (new-item -ItemType Directory -Path c:\temp\$((get-date -format yyyyMMdd)) )
```

Note that I'm using the top portion of the ISE to edit the file contents and I have saved the file as `dateFolder.ps1` in the `c:\temp` folder. To run the script, you can simply type the name of the file at prompt as follows (in the bottom portion of the ISE):



The screenshot shows the Windows PowerShell Integrated Scripting Environment (ISE) window. The title bar reads "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar with various icons is visible above the main area. The code editor pane contains a file named "datefolder.ps1" with the following content:

```
1 cd (new-item -ItemType Directory -Path c:\temp\$([get-date -format yyyyMMdd]) )
```

The output pane shows the command being run and its result:

```
PS C:\> C:\temp\datefolder.ps1
PS C:\temp\20150428> |
```

The status bar at the bottom indicates "Ln 3 Col 22" and "100%".

If you have the file loaded in the ISE, you can also use the **Run** button on the toolbar to run the current file.

It's possible that when you try to run this script, you will receive an error complaining about the execution policy, instead of seeing the current path being set to a new folder. To understand why this would have happened, we need to discuss the concept of execution policies.

Execution policy

Execution policy is a safety feature in PowerShell that enables the system to control which scripts are able to be run. I say *safety* instead of *security* because execution policy is trivial to circumvent. Execution policy are more like the safety of a gun, which prevents accidental discharge of the weapon. An execution policy is an attempt to prevent users from accidentally executing scripts.

Possible execution policy values include the following:

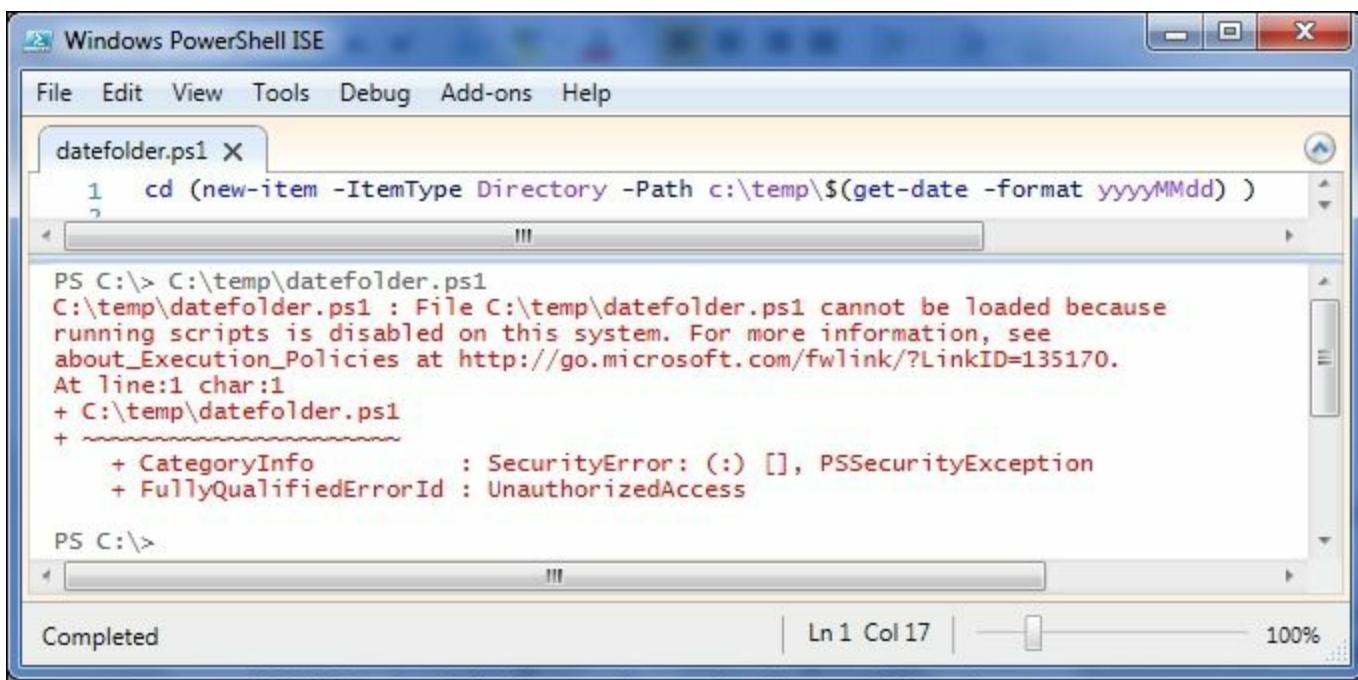
- Restricted
- AllSigned
- RemoteSigned
- Unrestricted

The **Restricted** setting means that the scripts cannot be run at all. **AllSigned** means that all scripts must be digitally signed to run.

Unrestricted says that any script will run. RemoteSigned is a middle-ground setting that says that any local scripts will run, but scripts from remote sources (like the Internet or untrusted UNC paths) need to be digitally signed.

Prior to Windows Server 2012R2, the default execution policy for all systems was Restricted, meaning that by default scripts were not allowed to run at all. With Server 2012R2, the default was changed to RemoteSigned.

Attempting to run a script when the execution policy does not permit it, results in an error such as the following:



The screenshot shows a Windows PowerShell ISE window. The title bar reads "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The code editor tab is titled "datefolder.ps1". The code in the editor is:1 cd (new-item -ItemType Directory -Path c:\temp\\$(\$get-date -format yyyyMMdd))

```
PS C:\> C:\temp\datefolder.ps1
C:\temp\datefolder.ps1 : File C:\temp\datefolder.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
+ C:\temp\datefolder.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: () [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
```

PS C:\>The status bar at the bottom left says "Completed", the cursor position is "Ln 1 Col 17", and the zoom level is "100%".

To see what your current execution policy is set to, use the `Get-ExecutionPolicy` cmdlet. To set the execution policy, you would use the `Set-ExecutionPolicy` cmdlet.

Tip

Important!

Since the execution policy is a system setting, changing it requires you to run an elevated session. Attempting to change the execution policy

from a user session, will result in an "Access Denied" error writing to the registry.

The following figure shows the results of running the script after the execution policy has been set to an appropriate level:

A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The window title is "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View (which is selected), Tools, Debug, Add-ons, and Help. The main pane displays a PowerShell script. The first few lines show an attempt to run a script named "datefolder.ps1" from the "C:\temp" directory. The output indicates that the script cannot be loaded because running scripts is disabled on this system. It provides a link to "about_Execution_Policies" for more information. The error message is in red. Below this, the command "Set-ExecutionPolicy RemoteSigned" is run, followed by another attempt to run "datefolder.ps1", which succeeds without any errors. The status bar at the bottom right shows "Ln 15 Col 22" and "100%".

```
PS C:\temp> C:\temp\datefolder.ps1
C:\temp\datefolder.ps1 : File C:\temp\datefolder.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
+ C:\temp\datefolder.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: () [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

PS C:\temp> Set-ExecutionPolicy RemoteSigned

PS C:\temp> C:\temp\datefolder.ps1

PS C:\temp\20150721>
```

In my experience, the `RemoteSigned` setting is most practical. However, in a secure environment such as a production data center, I can easily see that using an `AllSigned` policy could make sense.

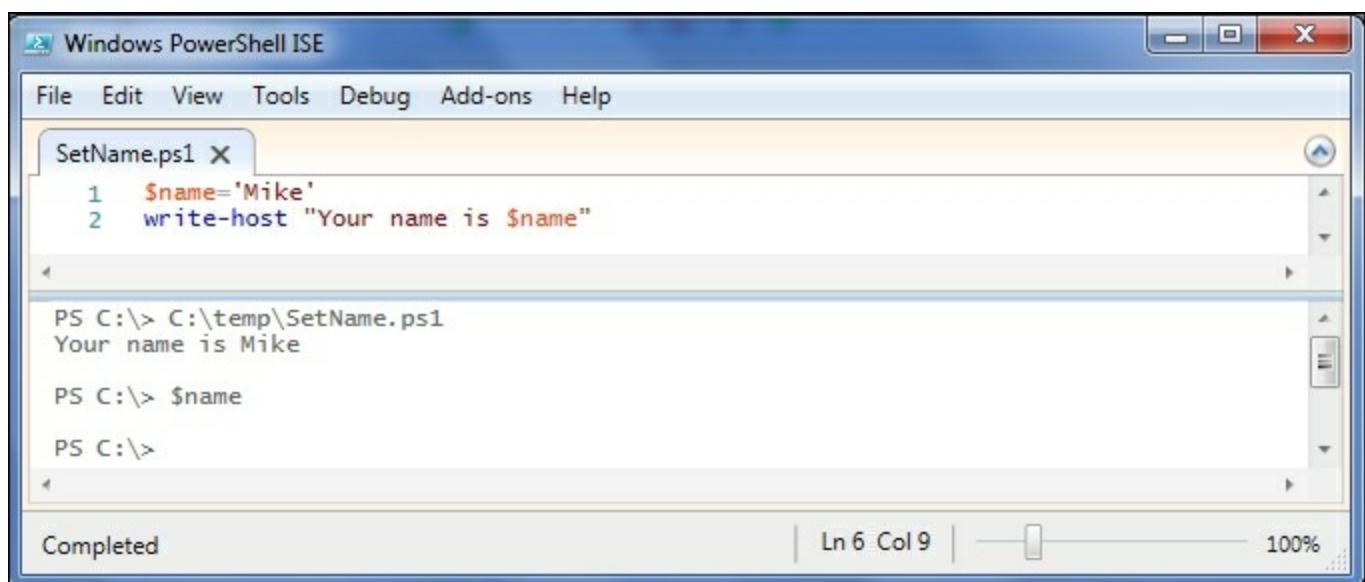
Types of scripts

The PowerShell community often distinguishes between two different kinds of scripts. Some scripts, called **controller scripts**, call other cmdlets, scripts, and functions to complete a process. Controller scripts sometimes do more than one thing, but they are not meant for reuse, except perhaps as a step in a larger controller script. You might expect to find a controller script scheduled to run in the task scheduler. **Tool scripts**, on the other hand, are more focused, perform a single action, and are intended specifically to be reused. A tool script would never run on its own, let alone be scheduled. There aren't any technical differences

between the two kinds of scripts, but there are things that would be considered best practices for one type that wouldn't be appropriate for the other. I find the distinction between the tools and controllers useful. When you write a script, think about whether it is a standalone process (controller), or it is something that you will use as a component in solving problems (tool).

Scopes and scripts

When you run a script, the PowerShell engine creates what is called a **scope** for the execution. If your script creates or modifies variables or functions (which we will discuss in later chapters), these changes will be created in the script's scope and will be removed at the end of the execution of the script. In the next chapter on PowerShell functions, we will see an important consequence of this, but for now we should just know that the things that are done in the script don't leave the script. As an example, here is a script that sets a variable to my name and outputs the variable to verify that it was correctly set. After executing the script, checking the variable shows that it is no longer set:



The screenshot shows the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A tab labeled "SetName.ps1" is open. The code in the editor is:

```
1 $name='Mike'
2 write-host "Your name is $name"
```

Below the editor, the PowerShell console window shows the following output:

```
PS C:\> C:\temp\SetName.ps1
Your name is Mike

PS C:\> $name

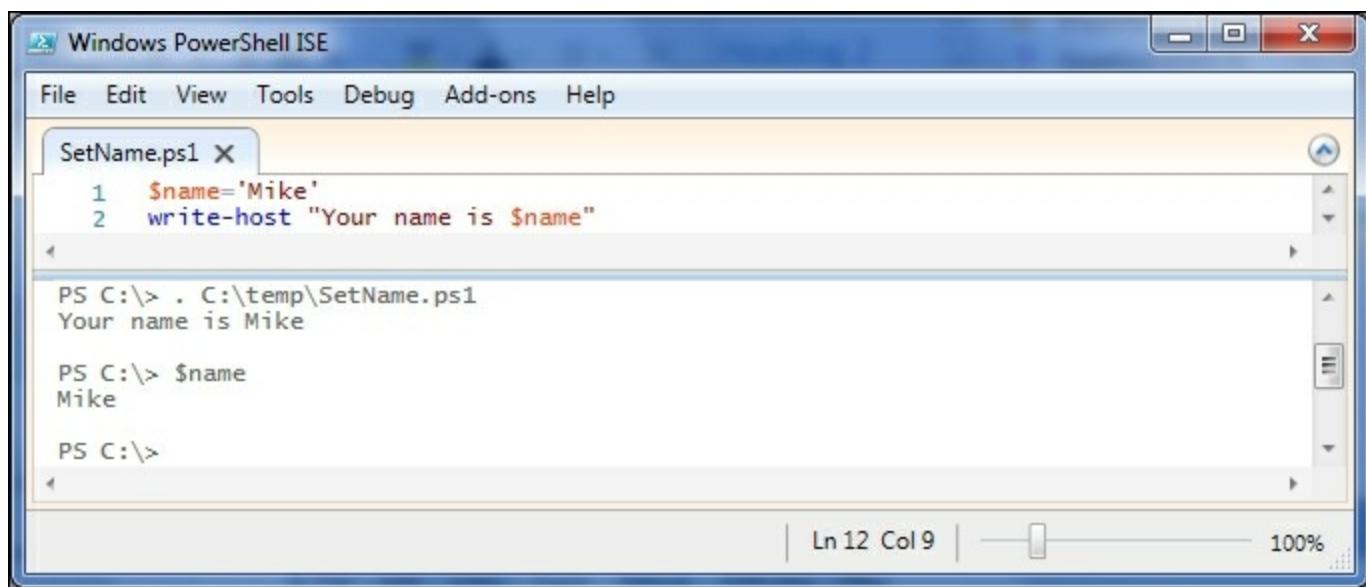
PS C:\>
```

The status bar at the bottom indicates "Completed" and "Ln 6 Col 9".

If we need the things in the script to remain after its execution, PowerShell provides the ability to **dot-source** the file rather than just executing it. Dot-sourcing means to execute something, but to do it in

the current scope and not create a child scope for it. The effect of dot-sourcing a script is the same as executing the commands in the script at the command-line—that is, all the effects of the script remain. To dot-source something, you use a period (a dot) and then a space, and then the name of the script.

If we dot-source the script from the previous example (instead of simply executing it), we can see that the \$name variable retains the value that was set in the script:



The screenshot shows the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A tab labeled "SetName.ps1" is open. The code editor contains the following script:

```
1 $name='Mike'
2 write-host "Your name is $name"
```

Below the code editor is a command prompt window. It shows the command ". C:\temp\SetName.ps1" being run, followed by the output "Your name is Mike". Then, the command "\$name" is run, resulting in the output "Mike". The status bar at the bottom indicates "Ln 12 Col 9" and "100%".

Parameters add flexibility

Executing a series of commands can be useful, but often you will want to execute the scripts and also provide some input—specific details about the execution of the script. One way to do this is to use **parameters**. Parameters are a way for you to provide the values that you want to be used in a script at execution time. For example, you might find that when you create a new directory, you immediately want to change to that directory. We started the chapter with a similar script that created a folder named with the current date, but because the script determined the folder name, we didn't need to provide any input. For this script, we don't know when we write the script what the name of the folder should be, but we will when we run it.

Parameters for scripts are specified in a `Param()` statement at the beginning of the script. The names of the parameters are variables and start with a dollar sign (\$). You aren't limited to a single parameter, either. You can include multiple parameters, separating them with commas. We will spend a lot more time talking about parameters when we talk about functions, but the following screenshot should illustrate how to use parameters in scripts:

Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

SetName.ps1 mdAndGo.ps1 X

```
1 Param($path)
2 new-item -ItemType Directory -Name $path
3 cd $path
```

PS C:\> C:\temp\mdAndGo Mike

Directory: C:\

Mode	LastWriteTime	Length	Name
d----	4/29/2015 7:30 PM		Mike

PS C:\Mike>

Completed | Ln 13 Col 13 | 100%

The screenshot shows a Windows PowerShell ISE window with two tabs: "SetName.ps1" and "mdAndGo.ps1". The "mdAndGo.ps1" tab is currently selected. The code in the editor pane is:

```
1 Param($path)
2 new-item -ItemType Directory -Name $path
3 cd $path
```

Below the editor, the PowerShell prompt PS C:\> is followed by the command C:\temp\mdAndGo Mike. The output pane shows the directory structure:

Directory: C:\

Mode	LastWriteTime	Length	Name
d----	4/29/2015 7:30 PM		Mike

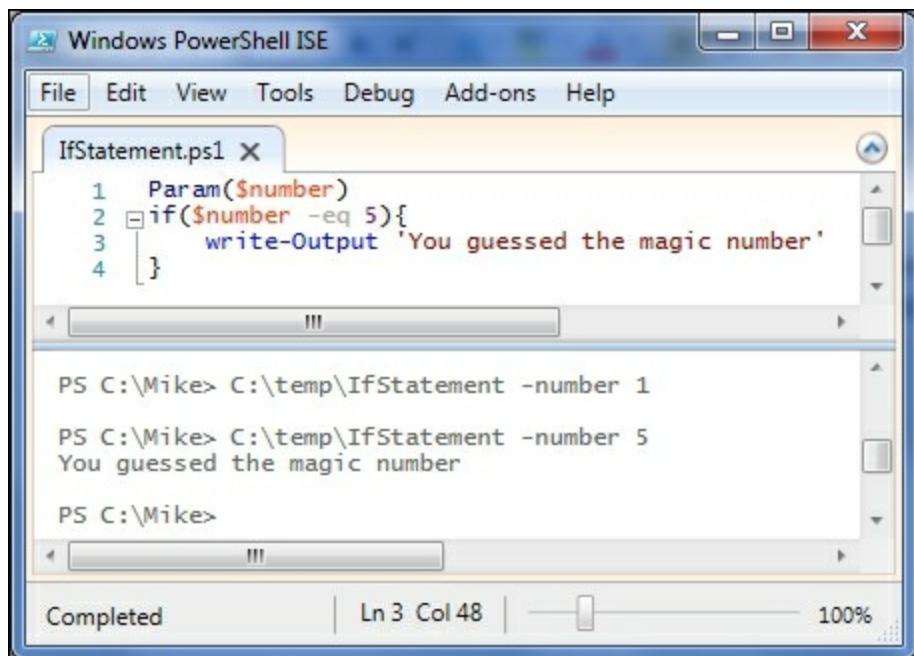
The current working directory is now C:\Mike, as indicated by the prompt PS C:\Mike>. The status bar at the bottom shows "Completed" and "Ln 13 Col 13" along with a zoom slider and "100%".

Adding some logic

PowerShell would not be very useful if scripts were limited to running one command after another without any logic. As expected from a modern language, PowerShell includes a full complement of control structures to allow you to vary the execution in just about any way you'd like. We will briefly introduce the most common control structures.

Conditional logic (IF)

The most simple logic statement is the `If` statement. The `If` statement allows you to have some code executed, if and only if a particular condition is met. In its simplest form, the `If` statement includes a condition in parentheses and a scriptblock to be executed when the condition is true:



The screenshot shows the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A tab labeled "IfStatement.ps1" is selected. The code editor contains the following PowerShell script:

```
1 Param($number)
2 if($number -eq 5){
3     write-output 'You guessed the magic number'
4 }
```

Below the code editor, the PowerShell console window shows the script's output:

```
PS C:\Mike> C:\temp\IfStatement -number 1
PS C:\Mike> C:\temp\IfStatement -number 5
You guessed the magic number
PS C:\Mike>
```

The status bar at the bottom indicates "Completed" and "Ln 3 Col 48".

If there is another condition that should be executed if the first condition is false, you can include an `Else` clause:

The screenshot shows the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A tab labeled "IfElse.ps1" is open. The code in the editor is:

```
1 Param($number)
2 if($number -eq 5){
3     write-output 'You guessed the magic number'
4 } else {
5     write-output 'You did not guess it, you lose'
6 }
```

Below the editor, the output pane shows two runs of the script:

```
PS C:\Mike> C:\temp\Ifelse -number 5
You guessed the magic number

PS C:\Mike> C:\temp\Ifelse -number 1
You did not guess it, you lose
```

The status bar at the bottom indicates "Completed" and "Ln 24 Col 13" with a zoom level of "100%".

If you have multiple conditions to test, there is an `ElseIf` clause as well. Here's an example of its use:

The screenshot shows the Windows PowerShell ISE interface. The title bar says "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A tab labeled "IfElseIf.ps1" is open. The code in the editor is:

```
1 $weekDay=Get-Date | select-Object -ExpandProperty DayOfWeek
2 if($weekDay -eq 'Saturday'){
3     write-host '2 days of weekend left'
4 } elseif ($weekDay -eq 'Sunday'){
5     write-host '1 day of weekend left'
6 } else {
7     write-host 'Have to go to work today!'
8 }
```

Below the editor, the output pane shows the result of running the script:

```
PS > .\IfElseIf.ps1
Have to go to work today!
```

The status bar at the bottom indicates "Ln 4 Col 6" with a zoom level of "100%".

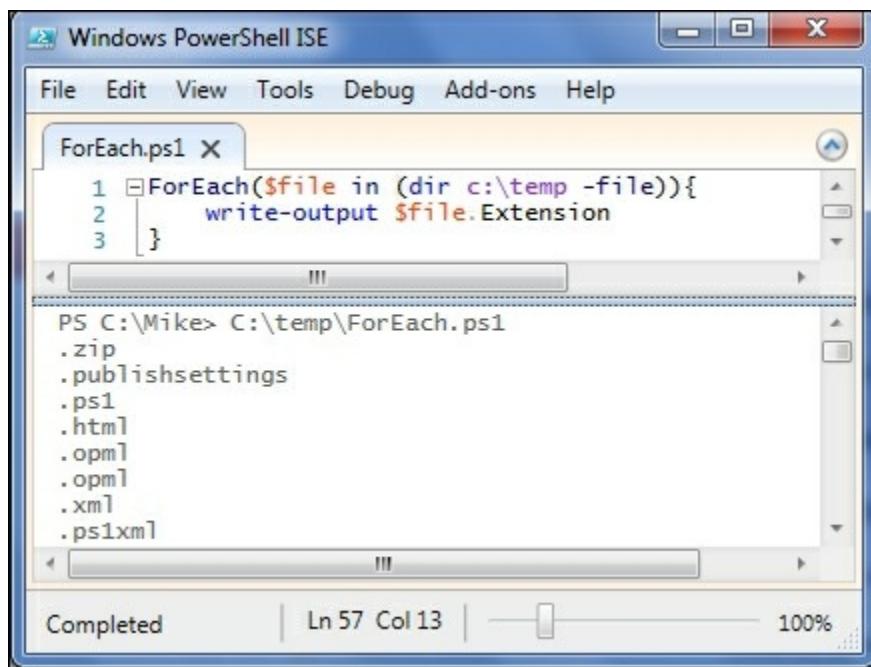
Looping logic

The most common way to loop in PowerShell is the `ForEach` statement.

ForEach causes a scriptblock to be executed for each item in a collection. The `ForEach` statement lets you name the placeholder that is used to refer to the individual items in the collection. The syntax for the `ForEach` statement looks as follows:

```
ForEach ($placeholder in $collection) {  
    #Code goes here  
}
```

As an example, we can display the extensions of all of the files in a folder using `ForEach`:



```
Windows PowerShell ISE  
File Edit View Tools Debug Add-ons Help  
ForEach.ps1 X  
1 ForEach($file in (dir c:\temp -file)){  
2     write-output $file.Extension  
3 }  
PS C:\Mike> C:\temp\ForEach.ps1  
.zip  
.publishsettings  
.ps1  
.html  
.opml  
.opml  
.xml  
.ps1xml  
Completed | Ln 57 Col 13 | 100%
```

If you remember from [Chapter 4, Life on the Assembly Line](#) there is also a `ForEach-Object` cmdlet that works in a pipeline and executes a scriptblock for each item in the pipeline. To make things even more confusing, `ForEach-Object` has an alias called `ForEach`. It's easy to get confused between the `ForEach` statement and the `ForEach-Object` cmdlet, but remember that the `ForEach` statement does not work in a pipeline and lets you name the placeholder, so if it's in a pipeline, it has to be the `ForEach-Object` cmdlet.

More logic

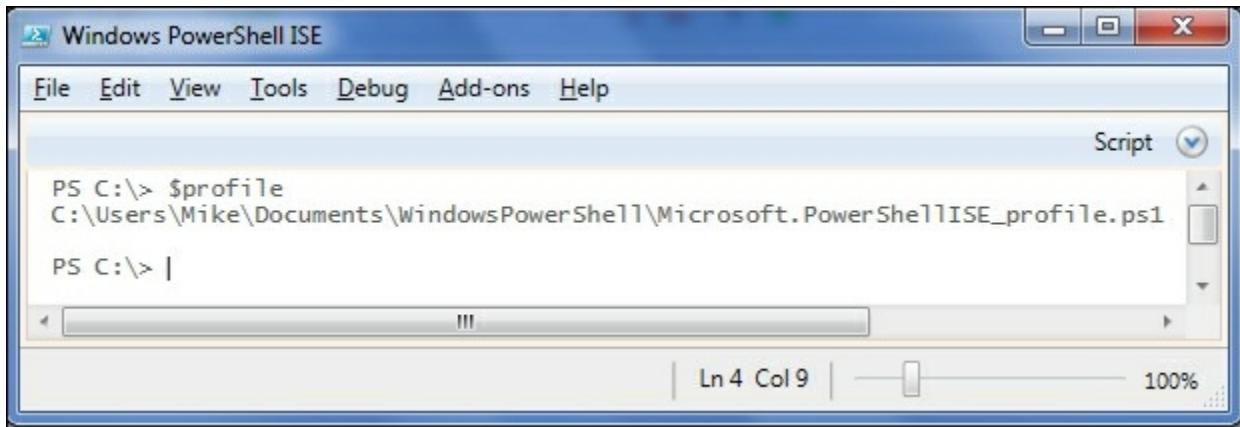
There are other statements that can be used to structure scripts, but `If` and `ForEach` are by far the most common. If you need to use others, they are well documented in PowerShell's awesome help content.

For quick reference, some of the other statements include the following:

Statement	Purpose
<code>While</code>	Executes a scriptblock as long as a condition is true
<code>Do...Until</code>	Executes a scriptblock until a condition is true
<code>For</code>	Executes a scriptblock for each number in a range (like 1 to 10)

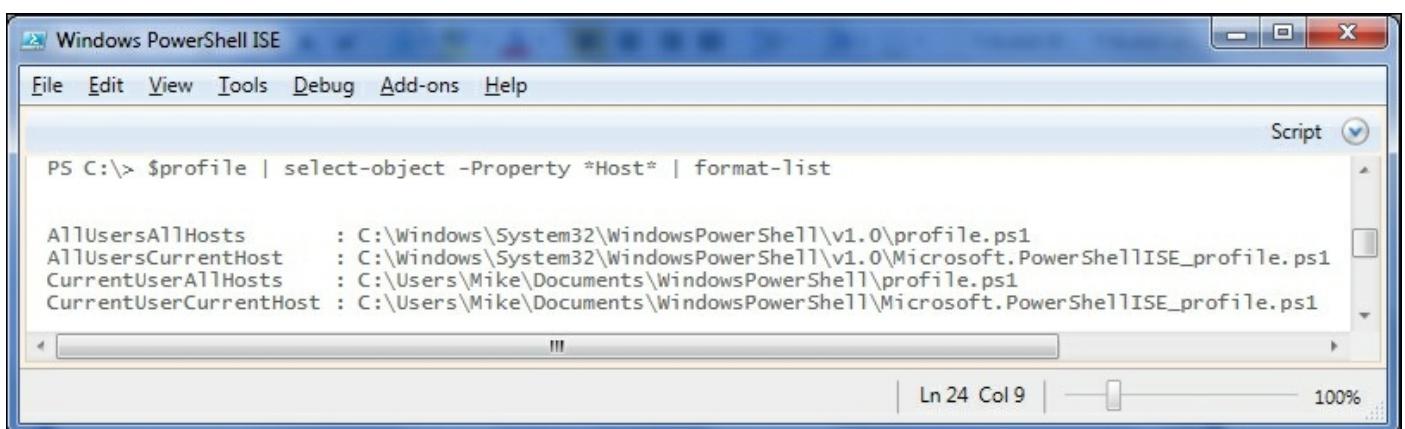
Profiles

Profiles are special scripts that PowerShell hosts run automatically when they start a new session. Each host looks in four different locations for profile scripts, running any of them that it finds. To find where these locations are, for the host you're in, look at the `$profile` automatic variable:



A screenshot of the Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar with a "Script" dropdown is visible. The main pane shows the command PS C:\> \$profile followed by its output: C:\Users\Mike\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1. Below that is another command PS C:\> |.

The first thing you'll notice is that there's only one path listed. To see all of them, you have to look at the extra properties that have been spliced onto `$profile`:

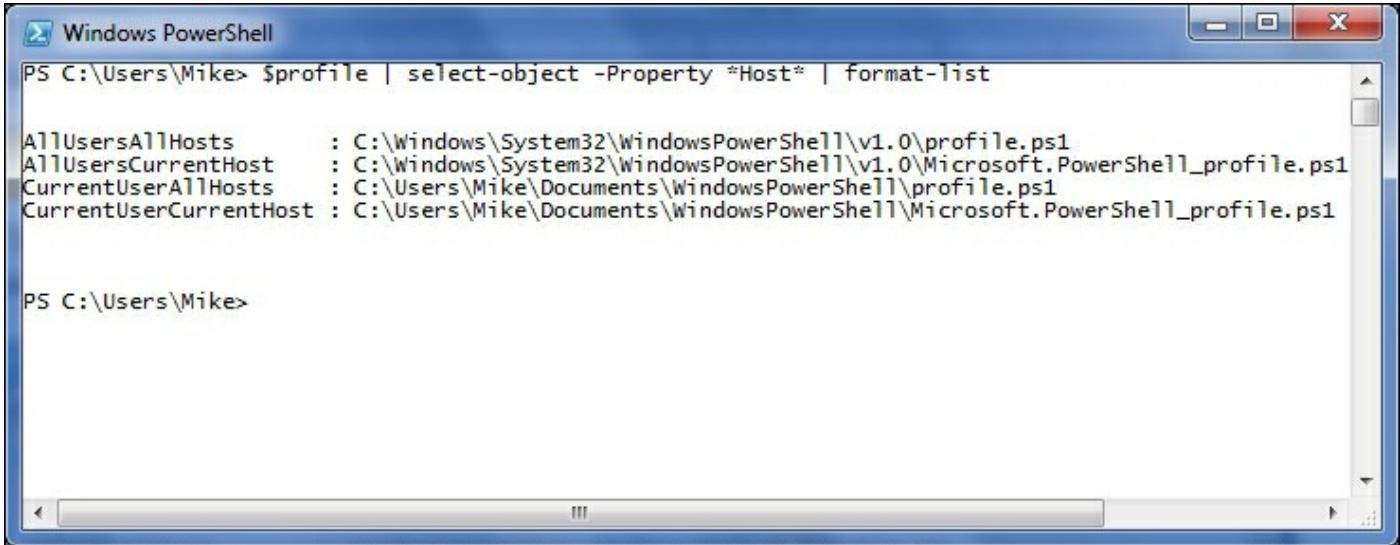


A screenshot of the Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A toolbar with a "Script" dropdown is visible. The main pane shows the command PS C:\> \$profile | select-object -Property *Host* | format-list followed by its output:

Property	Value
AllUsersAllHosts	C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost	C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1
CurrentUserAllHosts	C:\Users\Mike\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost	C:\Users\Mike\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1

The names of these properties should give you a good idea when each of them is used. The `AllHosts` profiles are used by all the hosts. The `AllUsers` profiles are used by all the users. The `CurrentUser` profiles are

used by the host you're in (console or ISE). The `CurrentUser` profiles are specific to the user running this PowerShell session. It might be helpful to see the output of the same command from the PowerShell console instead of the ISE. The `AllHosts` profiles are the same, but the `CurrentHost` profiles are different, specific to that host.



A screenshot of a Windows PowerShell window titled "Windows PowerShell". The window shows the command PS C:\Users\Mike> \$profile | select-object -Property *Host* | format-list being run. The output displays four entries: AllUsersAllHosts, AllUsersCurrentHost, CurrentUserAllHosts, and CurrentUserCurrentHost, each with its corresponding path. The window has a standard blue title bar and a scroll bar on the right side.

```
PS C:\Users\Mike> $profile | select-object -Property *Host* | format-list
AllUsersAllHosts      : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost   : C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts   : C:\Users\Mike\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost: C:\Users\Mike\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1

PS C:\Users\Mike>
```

You can put any code you want in your profile. At this point in Module 1, we don't have a lot of things that would be useful, but we could at least change the directory to a more useful location than `$PSHome`. Since profiles are scripts, they are subject to the execution policy limitations. This means if your execution policy is still set to `Restricted`, the host won't be able to run your profile script, even if it is in the right place. Similarly, if your execution policy is `AllSigned`, you would need to digitally sign your profile scripts.

Summary

Scripts offer a useful way to combine commands and reuse them later. In this chapter, we saw how to create and execute scripts. We learned about using parameters to make scripts more flexible and how to incorporate logic into a script using a few control structures.

In the next chapter, we will look at another way to package code for reuse: namely, functions.

For further reading

- Get-Help about_scripts
- Get-Help about_execution_policies
- Get-Help Set-ExecutionPolicy
- Get-Help about_scopes
- Get-Help about_parameters
- Get-Help about_profiles
- Get-Help about_if
- Get-Help about_foreach
- Get-Help about_switch
- Get-Help about_do
- Get-Help about_while
- Get-Help about_switch
- Get-Help about_for
- Get-Help about_break
- Get-Help about_continue

Chapter 7. Functions

In the previous chapter, we learned about packaging a set of commands in a script. In this chapter, we will look at using functions to provide the same kind of containment. We will discuss the differences between functions and scripts, and drill down more into parameters. The specific topics covered in this chapter include the following:

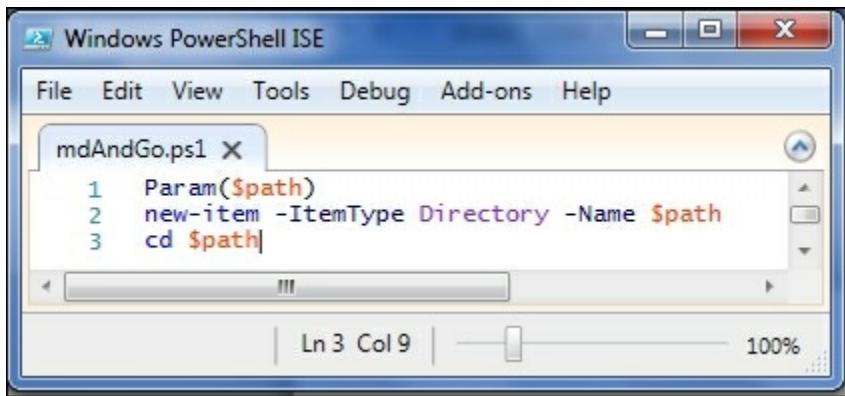
- Defining functions
- Comparing scripts and functions
- Executing functions
- Comment-based help
- Function output
- Parameters
- Default values

Another kind of container

Scripts are a simple way to reuse your commands. Scripts are file-based, so there is an inherent limitation—you can only have one script per file. This may not seem like a problem, but when you have hundreds or even thousands of pieces of code, you may want to be able to group them together and not have so many files on the disk.

Comparing scripts and functions

The primary difference between a function and a script is that a script is a file, whereas a function is contained within a file. Consider the `MdAndGo.ps1` script from [Chapter 6, Scripts](#):



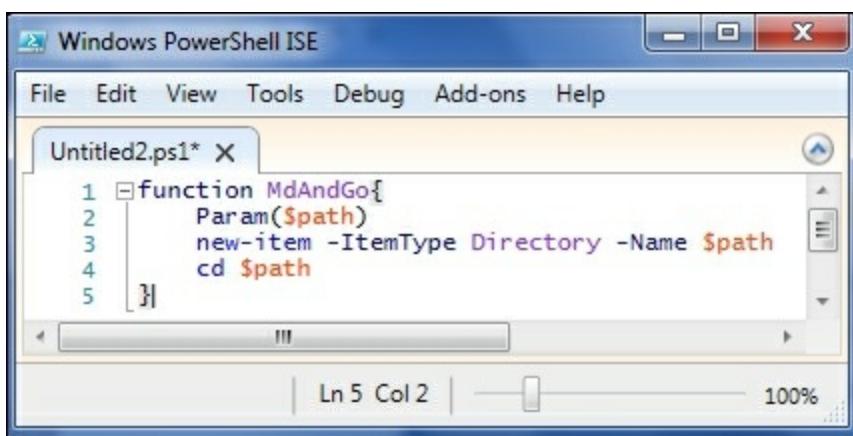
```
Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

mdAndGo.ps1 X
1 Param($path)
2 new-item -ItemType Directory -Name $path
3 cd $path

Ln 3 Col 9 | 100% |
```

As a function, this could be written like this:



```
Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Untitled2.ps1* X
1 function MdAndGo{
2     Param($path)
3     new-item -ItemType Directory -Name $path
4     cd $path
5 }

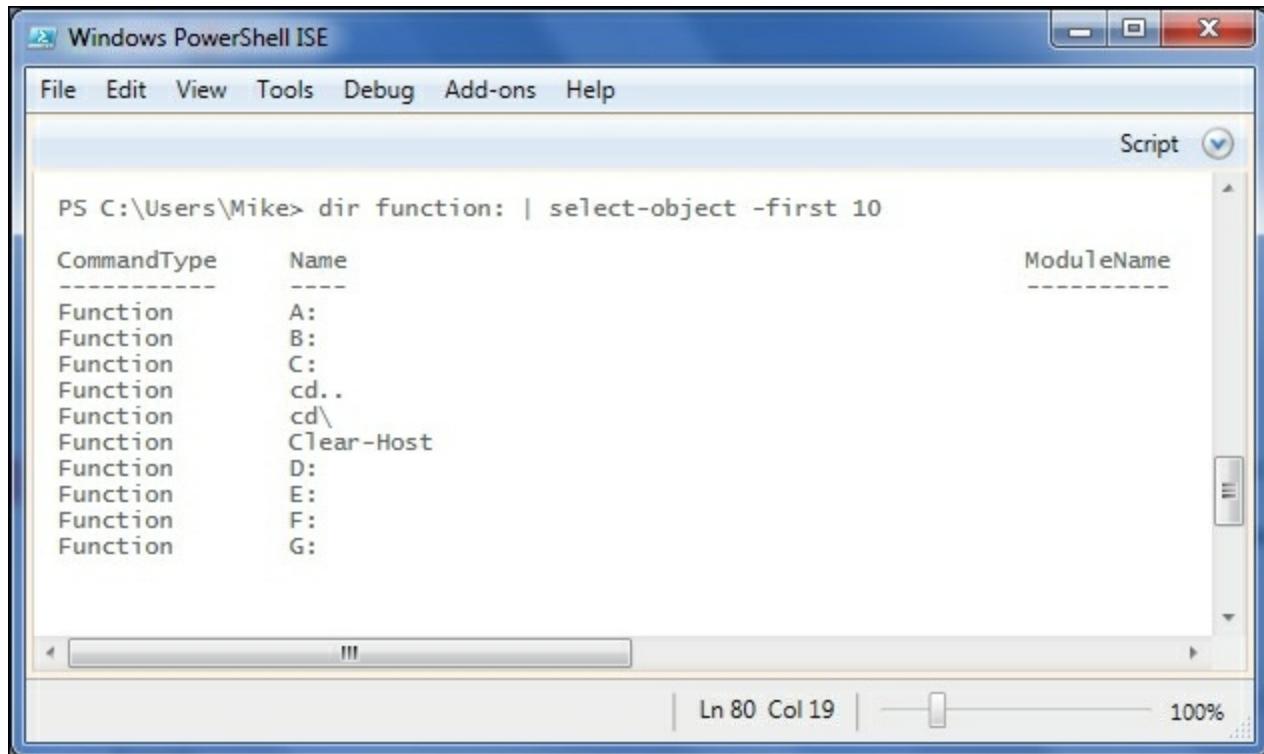
Ln 5 Col 2 | 100% |
```

You can see from this example that the functions are defined using the `function` keyword. Comparing these two examples, you can also see that the body of the function (the scriptblock after the function name) is identical to the contents of the script. Practically speaking, anything you can do in a script, you can do in a function also. If this is true (it is true), why would we want to use functions?

The first reason is simple convenience. If I write a dozen bits of code pertaining to a particular subject area, I would want these bits of code to be in one place. If I use scripts, I can put all of these in the same folder, and this gives a bit of coherence. On the other hand, if I use functions, I can put all of these functions in the same file. Since they're all in the same file, I can edit the file all at once and make the changes to the functions more easily. Seeing the functions in the same file can help to

understand the relationships between the code, and into repeated code or possibilities for refactoring.

Second, functions are exposed in PowerShell via a PSDrive called `Function:`. I can see all of the functions that have been loaded into my current session using cmdlets such as `Get-ChildItem`, or its commonly used alias `dir`:



The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with a "Script" button is visible. The main pane displays the following command and its output:

```
PS C:\Users\Mike> dir function: | select-object -first 10
```

CommandType	Name	ModuleName
Function	A:	
Function	B:	
Function	C:	
Function	cd..	
Function	cd\	
Function	Clear-Host	
Function	D:	
Function	E:	
Function	F:	
Function	G:	

The status bar at the bottom shows "Ln 80 Col 19" and "100%".

Tip

PSProviders and PSDrives

PowerShell uses PSProviders and PSDrives to provide access to hierarchical data sources using the same cmdlets that you would use to access filesystems. The `Function:` drive, mentioned here, holds all the functions loaded into the current session. Other PSDrives include the registry drives (`HKLM:` and `HCKU:`), the `Variable:` drive, the `ENV:` drive of environment variables, and the `Cert:` drive that provides access to the certificate store.

Executing and calling functions

As seen in figure in the *Comparing scripts and functions* section, defining a function involves using the `function` keyword and surrounding the code you want with a scriptblock. Executing this function statement simply adds the function definition to the current PowerShell session. To subsequently call the function, you use the function name like you would a cmdlet name, and supply parameters and arguments just like you would for a cmdlet or script.

Tip

Storing the `MdAndGo` function in a file called `MdAndGo.ps1` can be confusing because we're using the same name for the two things. You dot-source the `ps1` file to load the function into the session, then you can execute this function. Dot-sourcing the file doesn't run the function. If we had written this as a script, on the other hand, we could have executed the logic of the script without dot-sourcing.

The screenshot shows the Windows PowerShell ISE interface. In the top-left pane, there is a code editor window titled "Untitled2.ps1*" containing the following PowerShell script:

```
function MdAndGo{
    Param($path)
    new-item -ItemType Directory -Name $path
    cd $path}
```

In the bottom-right pane, the help output for the "MdAndGo" cmdlet is displayed:

```
PS C:\temp> get-help MdAndGo

NAME
    MdAndGo

SYNTAX
    MdAndGo [[-path] <Object>]

ALIASES
    None

REMARKS
    None
```

The status bar at the bottom indicates "Completed" and "Ln 20 Col 13".

Tip

Warning!

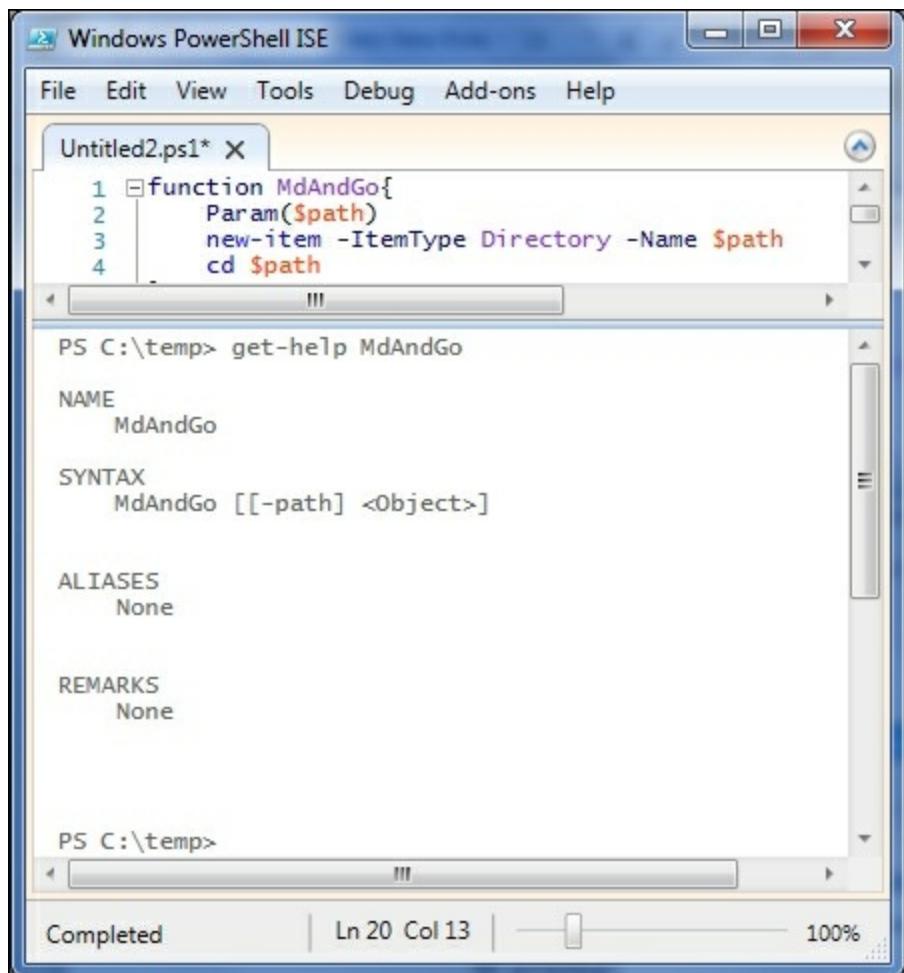
A common mistake of new PowerShell users is to call a function using parentheses to surround the parameters and commas to separate them. When you do this, PowerShell thinks you are creating an array and passes the array as an argument to the first parameter.

Naming conventions

Looking at the list of functions in the Function: drive, you will see some odd ones, such as `CD..`, `CD\`, and `A:.` These are created to help the command-line experience and are not typical functions. The names of these functions, though, show an interesting difference between functions and scripts, since it should be clear that you can't have a file named `A:.ps1`. Function naming, in general, should follow the same naming convention as cmdlets, namely Verb-Noun, where the verb is from the verb list returned by `Get-Verb` and the noun is a singular noun that consistently identifies what kinds of things are being referred to. Our `MdAndGo` function, for example, should have been named according to this standard. One reasonable name would be `Set-NewFolderLocation`.

Comment-based help

Simply defining a function creates a very basic help entry. Here's the default help for the `MdAndGo` function that we wrote earlier in the chapter:



The screenshot shows the Windows PowerShell ISE interface. In the top pane, there is a code editor window titled "Untitled2.ps1*" containing the following PowerShell script:

```
1 function MdAndGo{
2     Param($path)
3     new-item -ItemType Directory -Name $path
4     cd $path
```

In the bottom pane, the command `get-help MdAndGo` is run, and the resulting help output is displayed:

```
PS C:\temp> get-help MdAndGo

NAME
    MdAndGo

SYNTAX
    MdAndGo [[-path] <Object>]

ALIASES
    None

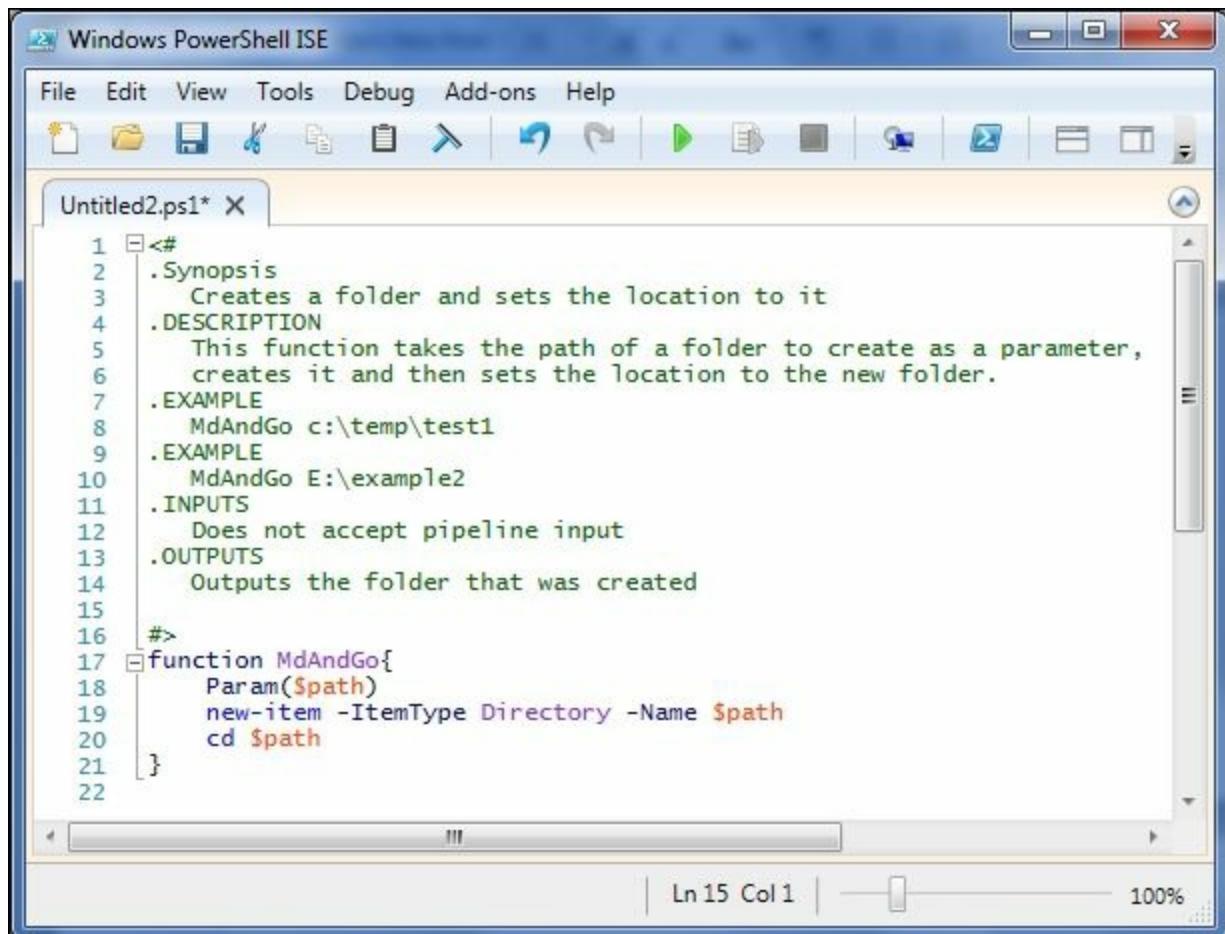
REMARKS
    None
```

The status bar at the bottom of the ISE window shows "Completed" and "Ln 20 Col 13".

We haven't given PowerShell enough to build a useful help entry, but fortunately for us there is a simple way to include help content through comments. This feature in PowerShell is called comment-based help and is documented thoroughly in the `about_comment_based_help` help topic. For our purposes, we will add enough information to show that we're able to affect the help output, but we won't go into much detail.

With comment-based help, you will include a specially formatted comment immediately before the function. Keywords in the comment

indicate where the help system should get the text for different areas in the help topic. Here's what our function looks like with an appropriate comment:



The screenshot shows the Windows PowerShell ISE interface with a script file named "Untitled2.ps1". The code contains PowerShell comments (`.Synopsis`, `.DESCRIPTION`, `.EXAMPLE`, etc.) and a function definition:

```
1 <#
2 .Synopsis
3     Creates a folder and sets the location to it
4 .DESCRIPTION
5     This function takes the path of a folder to create as a parameter,
6     creates it and then sets the location to the new folder.
7 .EXAMPLE
8     MdAndGo c:\temp\test1
9 .EXAMPLE
10    MdAndGo E:\example2
11 .INPUTS
12     Does not accept pipeline input
13 .OUTPUTS
14     Outputs the folder that was created
15
16 #>
17 function MdAndGo{
18     Param($path)
19     new-item -ItemType Directory -Name $path
20     cd $path
21 }
22
```

After executing the file to create the function, we can test the comment-based help using `get-help MdAndGo`, just as before:

The screenshot shows a Windows PowerShell ISE window with the title bar "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The main content area displays the help output for the cmdlet "MdAndGo".

```
PS C:\temp> get-help MdAndGo

NAME
    MdAndGo

SYNOPSIS
    Creates a folder and sets the location to it

SYNTAX
    MdAndGo [[-path] <Object>] [<CommonParameters>]

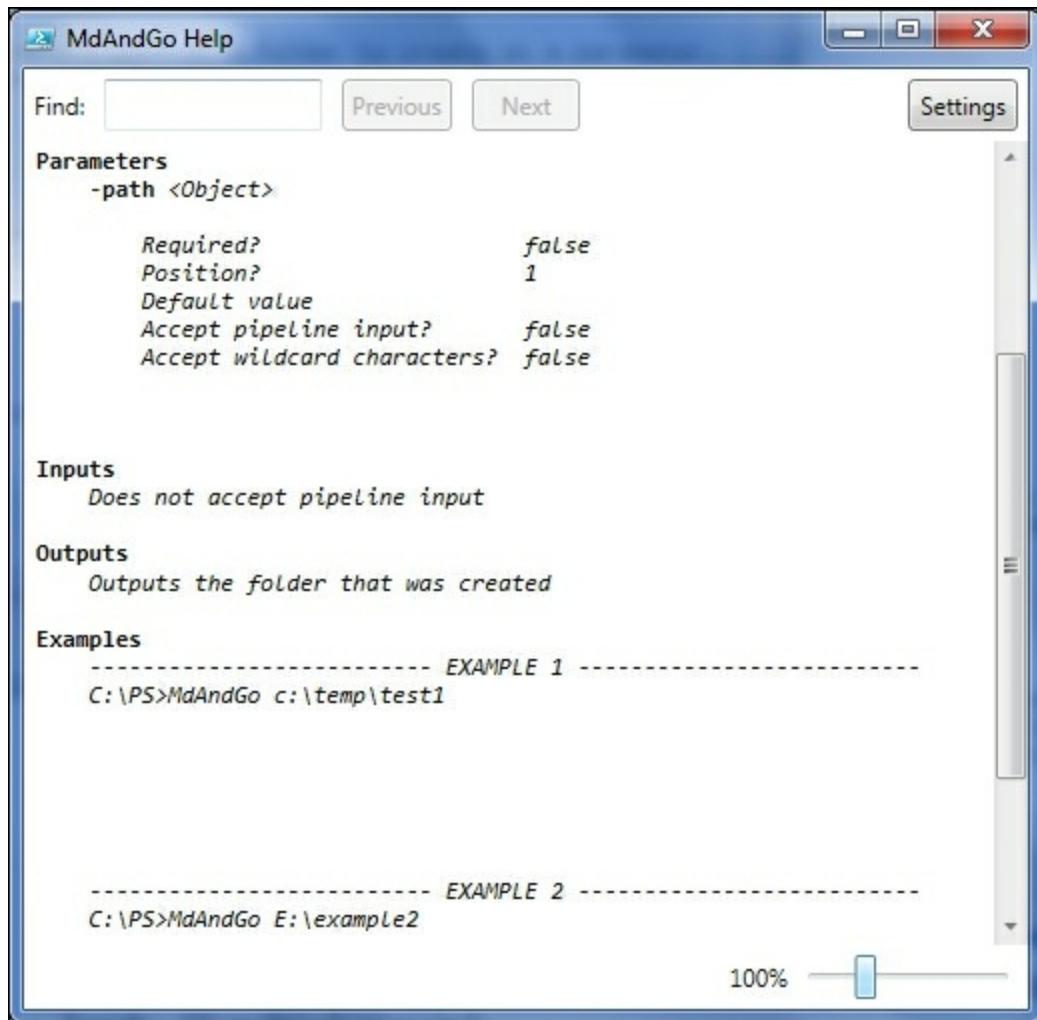
DESCRIPTION
    This function takes the path of a folder to create as a parameter,
    creates it and then sets the location to the new folder.

RELATED LINKS

REMARKS
    To see the examples, type: "get-help MdAndGo -examples".
    For more information, type: "get-help MdAndGo -detailed".
    For technical information, type: "get-help MdAndGo -full".
```

The status bar at the bottom shows "Completed" on the left, "Ln 90 Col 13" in the center, and "100%" on the right.

Since we included examples, we can either use the `-Examples` switch or use the really handy `-ShowWindow` switch:



Note that we didn't need to number our examples or include the `C:\PS>` prompt. These are added automatically by the help system.

Parameters revisited

Just like scripts, functions can take parameters. Remember that parameters are placeholders, which let you vary the execution of the function (or script) based on the value that is passed to the parameter (called the argument). The `MdAndGo` function that we wrote earlier had a single parameter, but you can use as many parameters as you want. For instance, if you had a process that copied items from one folder to another and you wanted to write a function to make sure that both the source and destination folders already exist, you could write a function like this:

```
Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

testPathsFunction.ps1 X
1 function test-paths{
2     Param([string]$path1,[string]$path2)
3     return (test-path $path1) -and (test-path $path2)
4 }
5

PS C:\temp> test-paths c:\temp D:\temp
True

PS C:\temp> |
```

It should be clear from this example that parameters in the `Param()` statement are separated by commas. Remember that you don't use commas to separate the arguments you pass when you call the function. Also, the `Param()` statement needs to be the first statement in the function.

Typed parameters

Something else you might have noticed in this example is that we have specified that both the parameters have to be strings. It's hard to test

this, since anything in the command-line is already a string. Let's write a function to output the average of two integers:

The screenshot shows the Windows PowerShell ISE interface. The script file 'get-average.ps1' contains the following code:

```
1 function get-average{
2     param([int]$a,[int]$b)
3     return ($a+$b)/2
4 }
```

The execution results show two runs of the function:

```
PS C:\temp> get-average 10 20
15

PS C:\temp> get-average orange apple
get-average : Cannot process argument transformation on parameter 'a'. Cannot
convert value "orange" to type "System.Int32". Error: "Input string was not in
a correct format."
At line:1 char:13
+ get-average orange apple
+ ~~~~~
+ CategoryInfo          : InvalidData: (:) [get-average], ParameterBinding
ArgumentTransformationException
+ FullyQualifiedErrorId : ParameterArgumentTransformationError, get-average
```

PS C:\temp>

The status bar at the bottom indicates 'Completed'.

Tip

Here, we've used `[int]` to specify that the `$a` and `$b` parameters are integers. Specifying a type allows us to make certain assumptions about the values in these variables, namely, that they are numeric and it makes sense to average them. Other common types used in PowerShell are `[String]` and `[DateTime]`.

The function works if you pass numeric arguments, but passing strings (such as `orange` and `apple`) causes an error. If you read the error message, you can see that PowerShell tried to figure out how to change `"orange"` into a number, but couldn't come up with a transformation that works. PowerShell has a very versatile parameter binding mechanism that attempts a number of methods to transform the arguments you pass into the type of parameters you have specified.

Switches

Besides using types like `string` and `int`, you can also use the `switch` type, which indicates a parameter that is either present or absent. We've seen switch parameters in cmdlets, such as the `-Recurse` switch parameter for `Get-ChildItem`. Remember that you don't specify an argument to a switch parameter. You either supply the parameter (such as `-Recurse`) or you don't. Switch parameters allow for easy on/off or true/false input. The parameter can be tested easily in an `if` statement like this:

The screenshot shows the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with icons for Save, Open, and Run is visible above the main area. The code editor window contains a script named "TestSwitchFunction.ps1" with the following content:

```
1 function test-switch{
2     param([switch]$MySwitch)
3     if ($MySwitch){
4         write-host 'You used the switch'
5     } else {
6         write-host 'You didn''t use the switch'
7     }
8 }
```

Below the code editor is a command prompt window showing the execution of the script:

```
PS C:\temp> test-switch
You didn't use the switch

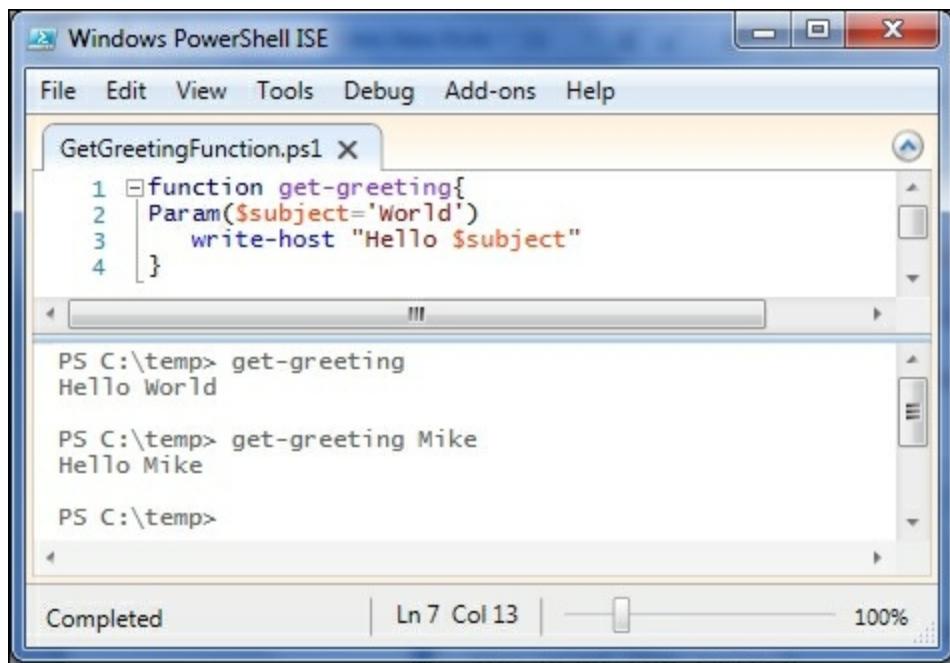
PS C:\temp> test-switch -MySwitch
You used the switch

PS C:\temp>
```

The status bar at the bottom shows "Completed" and "Ln 31 Col 13" along with a zoom slider set to 100%.

Default values for parameters

You can also supply default values for parameters in the `Param()` statement. If no argument is given for a parameter that has a default, the default value is used for this parameter. Here's a simple example:



The screenshot shows the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A tab labeled "GetGreetingFunction.ps1" is selected. The code editor contains the following PowerShell script:

```
1 function get-greeting{
2     Param($subject='World')
3         write-host "Hello $subject"
4 }
```

Below the code editor is a command-line window showing the execution of the script:

```
PS C:\temp> get-greeting
Hello World

PS C:\temp> get-greeting Mike
Hello Mike

PS C:\temp>
```

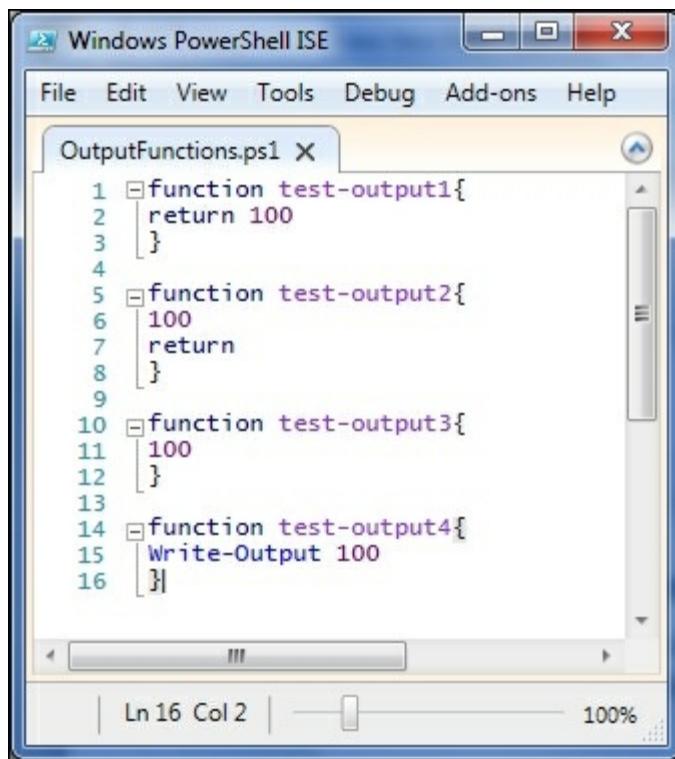
The status bar at the bottom indicates "Completed" and shows the cursor position as "Ln 7 Col 13" and a zoom level of "100%".

Using parameters is a powerful technique to make your functions applicable to more situations. This kind of flexibility is the key to writing reusable functions.

Output

The functions that I've shown so far in this chapter have either explicitly used the `return` keyword, or used `write-host` to output text to the console instead of returning the data. However, the mechanism that PowerShell uses for function output is much more complex for several reasons.

The first complication is that the `return` keyword is completely optional. Consider the following four functions:

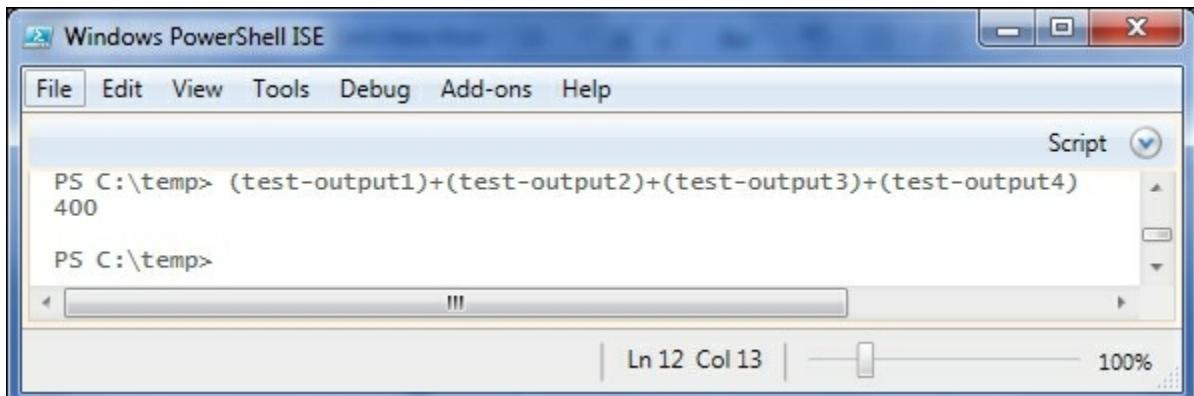


```
Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help
OutputFunctions.ps1 X

1 function test-output1{
2     return 100
3 }
4
5 function test-output2{
6     100
7     return
8 }
9
10 function test-output3{
11     100
12 }
13
14 function test-output4{
15     Write-Output 100
16 }
```

We can see that they all output the same value (100) by adding them all together:



The screenshot shows a Windows PowerShell ISE window titled "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with a "Script" button is visible. The main area displays the command PS C:\temp> (test-output1)+(test-output2)+(test-output3)+(test-output4) followed by the output 400. Below this, another line PS C:\temp> is shown. The status bar at the bottom indicates "Ln 12 Col 13" and "100%".

The reason that these functions do the same thing is that PowerShell uses the concept of streams to deal with data coming out of a function. Data that we consider "output" uses the output stream. Any value that isn't used somehow is written to the output stream, as is a value given in a return statement. A value can be used in many ways. Some common ways of using a value are:

- Assigning the value to a variable
- Using the value as a parameter to a function, script, or cmdlet
- Using the value in an expression
- Casting the value to [void] or piping it to Out-Null

Tip

[void] is a special type that indicates that there is no actual value. Casting a value to [void] is essentially throwing away the value.

The last function, Test-Output4, shows that we can explicitly write to the output stream using Write-Output.

Another interesting feature of the output stream is that the values are placed in the output stream immediately, rather than waiting for the function to finish before outputting everything. If you consider running DIR C:\ -recurse, you can imagine that you would probably want to see the output before the whole process is complete, so this makes sense.

Functions are not limited to a single output value, either. As I stated earlier, *any value* that isn't used is output from the function. Thus, the

following functions output two values and four values, respectively:

```
Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

MultipleOutput.ps1 X

1 function get-twovalues{
2     100
3     "Hello World"
4 }
5 function get-four{
6     100
7     "Hello world"
8     (get-date)
9     dir c:\temp | select-object -first 1 -Property FullName
10 }

PS C:\temp> get-twovalues
100
Hello World

PS C:\temp> get-four
100
Hello world
Tuesday, May 19, 2015 10:46:49 PM
FullName : C:\temp\PowerShellCookbook

Completed Ln 10 Col 2 100%
```

The output from these strange functions illustrates another property of PowerShell functions, namely, they can output more than one kind of object.

In practice, your functions will generally output one kind of object, but if you're not careful, you might accidentally forget to use a value somewhere and that value will become part of the output. This is a source of great confusion to PowerShell beginners, and can even cause people who have been using PowerShell to scratch their heads from time to time. The key is to be careful, and if in doubt, use `Get-Member` to see what kind of objects you get out of your function.

Summary

The functions in PowerShell are powerful and extremely useful to create flexible, reusable code. Parameters, including switch parameters, allow you to tailor the execution of a function to meet your needs. PowerShell functions behave differently from functions in many other languages, potentially outputting multiple objects of different types and at different times.

In the next chapter, we will look at PowerShell modules, a feature added in PowerShell 2.0 to help gather functions and scripts into libraries.

For further reading

- `Get-Help about_Functions`
- `Get-Help about_Parameters`
- `Get-Help about_Return`
- `Get-Help about_Comment_Based_Help`
- `Get-Help about_Providers`
- `Get-Help about_Core_Commands`

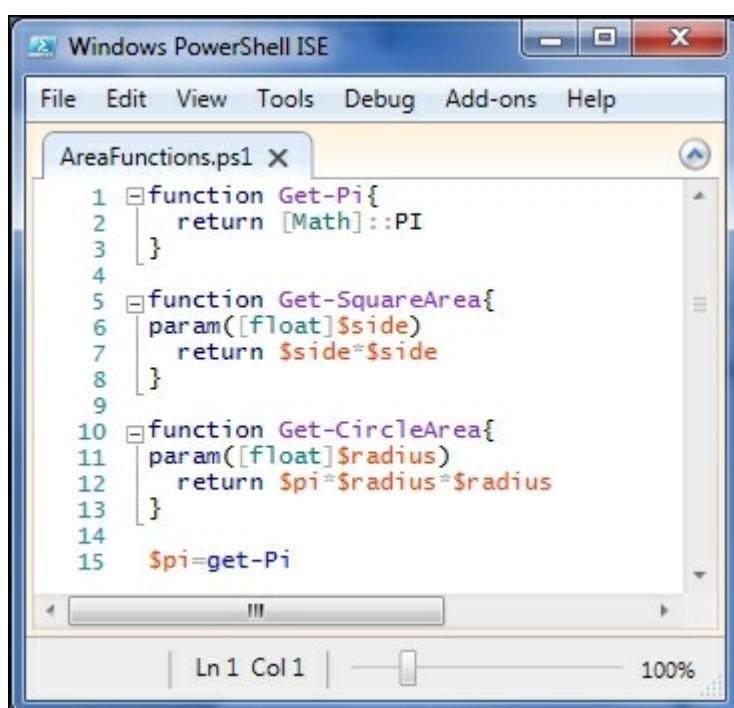
Chapter 8. Modules

In the last two chapters, we learned about bundling code into scripts and functions. In this chapter, we will learn about using modules to bundle multiple functions or script files that are full of functions into a single container. There's more to modules than simply grouping things together. Altogether, we will learn the following:

- What is a module?
- Where do modules live?
- Loading and unloading a module
- PowerShell module autoloading

Packaging functions

We learned in the last two chapters that you can put a bunch of functions in a script file, and then dot-source this file to get these functions into a session. Consider the following script, called `AreaFunctions.ps1`:



```
Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help
AreaFunctions.ps1 ×

1 function Get-Pi{
2     return [Math]::PI
3 }
4
5 function Get-SquareArea{
6     param([float]$side)
7     return $side*$side
8 }
9
10 function Get-CircleArea{
11     param([float]$radius)
12     return $pi*$radius*$radius
13 }
14
15 $pi=get-Pi

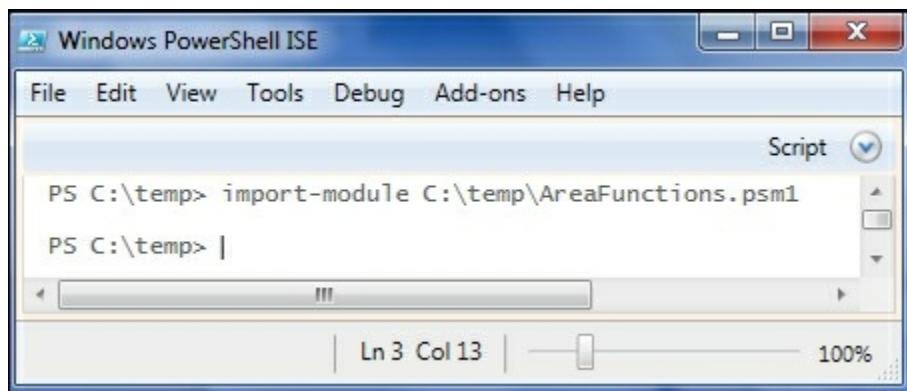
Ln 1 Col 1 | 100%
```

After dot-sourcing the script file, we can use the functions and access

the \$pi variable as well. If we use `get-command`, we will find the functions but they are no longer connected in any way. That is, it is difficult to see that these functions are part of a unit. There is no data retained in the function definitions that tells us where they came from. Since one of the important points of PowerShell is that it is discoverable, this situation isn't ideal. PowerShell 2.0 introduced the concept of **modules** to help tie these functions (among other things) together.

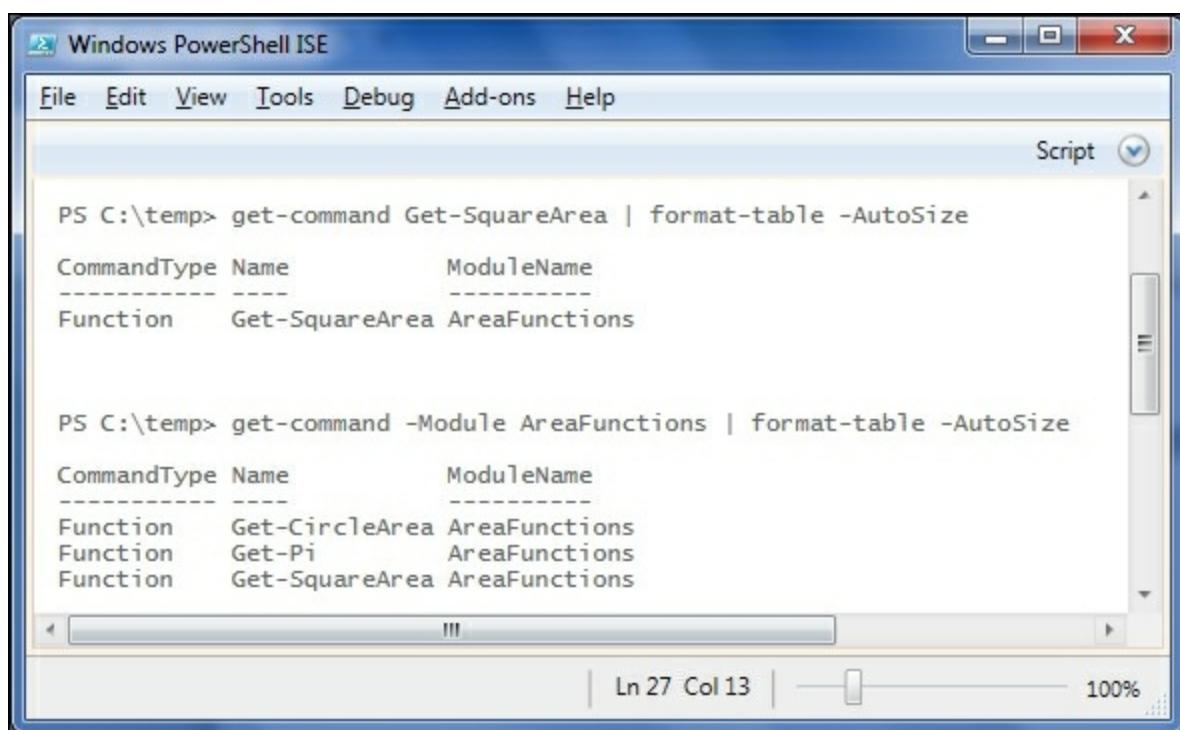
Script modules

To turn this into a module in the simplest way, we can just change the file extension to `.psm1`. Once we have done this, we can import the module by pointing directly to the `.psm1` file, as follows:



A screenshot of the Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu labeled "Script" is open. The main pane shows the command "PS C:\temp> import-module C:\temp\AreaFunctions.psm1" followed by a blank line. The status bar at the bottom indicates "Ln 3 Col 13" and "100%".

Similar to when we dot-sourced the script file, here we also don't see any real indication that the `import-module` cmdlet did anything. If we look at the output of `get-command` now, we can see that these functions are now listed as part of a module. We can, also, use the `-Module` parameter for `get-command` to show all the functions that were exported from the module:



A screenshot of the Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu labeled "Script" is open. The main pane shows two sets of command outputs. The first set is "PS C:\temp> get-command Get-SquareArea | format-table -AutoSize" followed by a table:

CommandType	Name	ModuleName
Function	Get-SquareArea	AreaFunctions

The second set is "PS C:\temp> get-command -Module AreaFunctions | format-table -AutoSize" followed by a table:

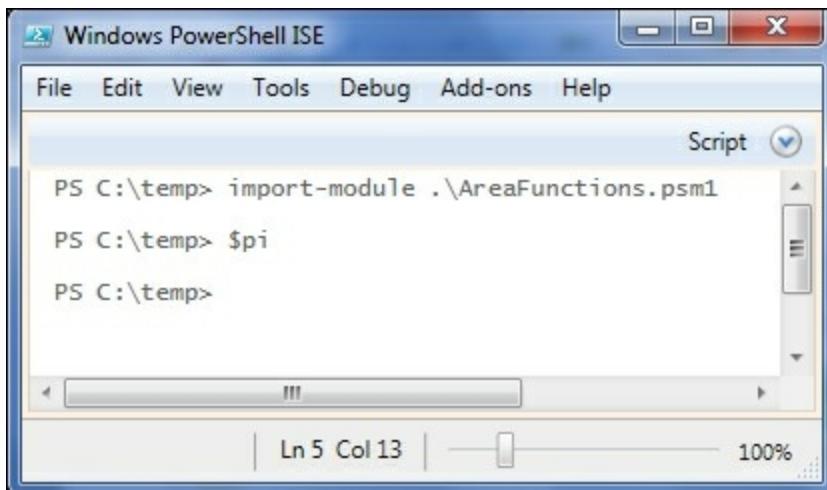
CommandType	Name	ModuleName
Function	Get-CircleArea	AreaFunctions
Function	Get-Pi	AreaFunctions
Function	Get-SquareArea	AreaFunctions

The status bar at the bottom indicates "Ln 27 Col 13" and "100%".

Now, these functions are much easier to discover, since we can see that they fit together somehow.

Module files with a `.psm1` extension are called script modules, for obvious reasons. One huge benefit of using script modules is that they are simple to create. Another reason for using script modules is that they provide the opportunity to hide elements that you don't want to be public.

In our module, the `Get-Pi` function is really just a helper function that is used to set the `$pi` variable (also, not super useful), which the `Get-CircleArea` function uses to perform its calculations. If we try to access `$pi`, we will find that the variable doesn't even exist. The reason is that by default, modules don't expose variables defined in them:



The screenshot shows a Windows PowerShell ISE window titled "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu labeled "Script" is open. The main pane displays a PowerShell session with the following commands:

```
PS C:\temp> import-module .\AreaFunctions.psm1
PS C:\temp> $pi
PS C:\temp>
```

The status bar at the bottom indicates "Ln 5 Col 13" and "100%".

We'll see in the next section how to remedy this, but first, let's see how we can keep the `Get-Pi` function from being exposed as well.

The Export-ModuleMember cmdlet

The key to controlling which parts of a module are visible is to know the default visibility and how to use the `Export-ModuleMember` cmdlet.

By default, a module does the following:

- Exports all the functions
- Hides all the variables
- Hides all the aliases

If you don't like one of these, you can override it using the `Export-ModuleMember` cmdlet. This cmdlet has three parameters called - `-Function`, `-Variable`, and `-Alias`, which let you list specifically which members (functions, variables, and aliases) you want to be exported. Since we are fine with the variable not being exported, and only want the two area functions exported, we can add this command at the end of the module to get the required configuration:

```
Export-ModuleMember -Function Get-SquareArea,Get-CircleArea
```

Tip

Warning!

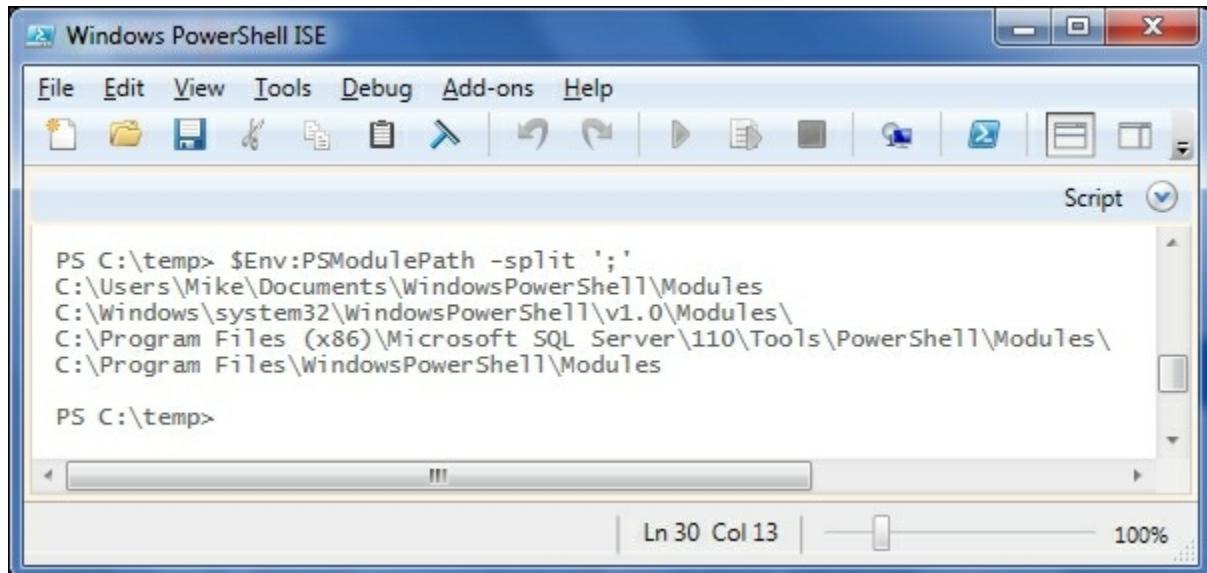
If you include any `Export-ModuleMember` commands in your module, you need to specify all the members you want to export. For instance, if we wanted to export the `Pi` variable, we would have added the following:

```
Export-ModuleMember -variable Pi
```

If this was the only `Export-ModuleMember` command in the module, none of the functions would be exported. Also, note that we don't include `$` when listing variable names in `Export-ModuleMember`.

Where do modules live?

PowerShell defines an environment variable called `PSMODULEPATH`, which contains a list of folders that it checks for modules. Viewing `PSMODULEPATH` is simplified by the `-split` operator to split the path wherever it sees a semicolon:



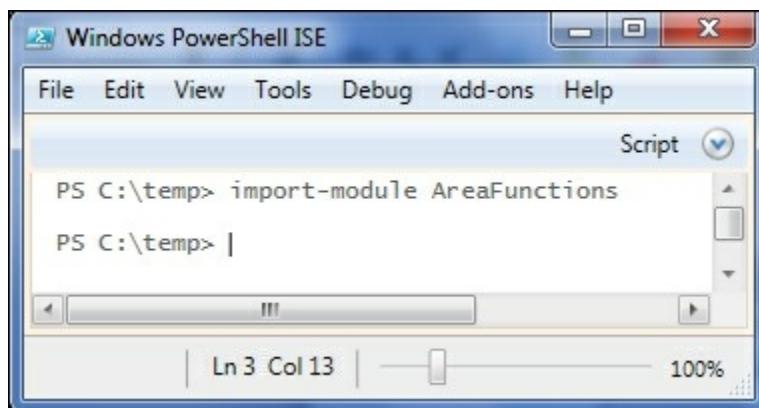
The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar has various icons for file operations like Open, Save, and Print. The main area is titled "Script" and contains the following PowerShell command and its output:

```
PS C:\temp> $Env:PSModulePath -split ';'
C:\Users\Mike\Documents\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
C:\Program Files (x86)\Microsoft SQL Server\110\Tools\PowerShell\Modules\
C:\Program Files\WindowsPowerShell\Modules
```

Below the command, the prompt "PS C:\temp>" is visible. At the bottom of the window, status bars show "Ln 30 Col 13" and "100%".

Here, you can see that there are four folders where PowerShell is looking for modules in this session. For it to find our `AreaFunctions` module, there needs to be a `AreaFunctions` subfolder in one of these locations with our `.psm1` file in it.

Once we have created a folder and placed the `AreaFunctions.psm1` file in it, we can import the module by name rather than by path:



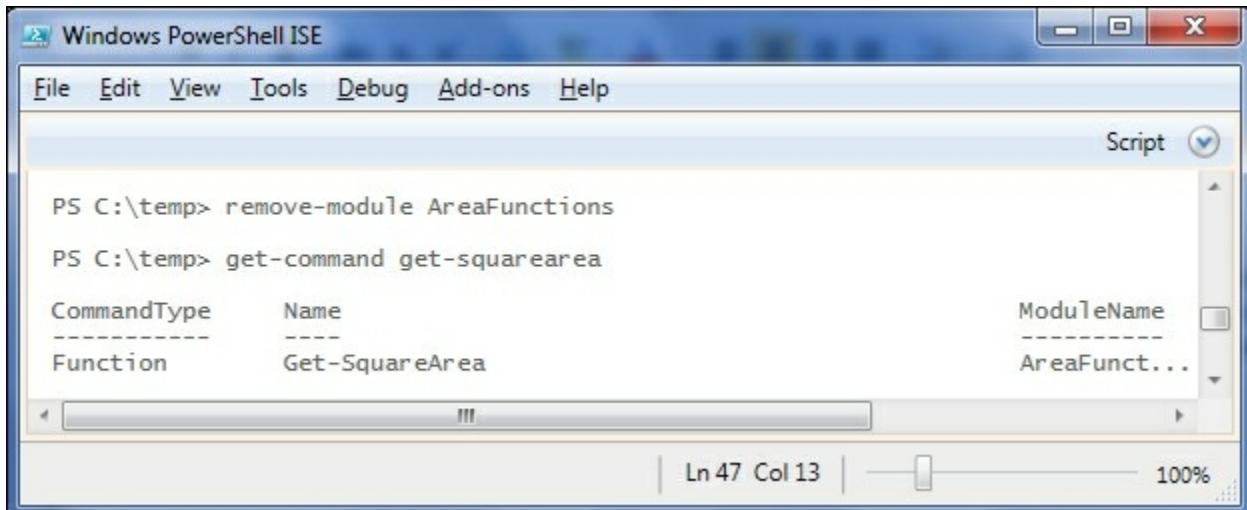
The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar has various icons for file operations like Open, Save, and Print. The main area is titled "Script" and contains the following PowerShell command:

```
PS C:\temp> import-module AreaFunctions
```

Below the command, the prompt "PS C:\temp>" is visible. At the bottom of the window, status bars show "Ln 3 Col 13" and "100%".

Removing a module

If you decide that you don't want the functions in a module included in your session for some reason, you can remove it using the `Remove-Module` cmdlet. Once the module is removed, you can try to check whether it was removed using the `Get-Command` cmdlet to look for one of the exported functions:



The screenshot shows a Windows PowerShell ISE window. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu is open under the View tab, showing 'Script' selected. The main pane displays the following PowerShell session:

```
PS C:\temp> remove-module AreaFunctions
PS C:\temp> get-command get-squarearea
CommandType      Name
-----          ----
Function        Get-SquareArea
ModuleName      AreaFunc...
```

The status bar at the bottom indicates 'Ln 47 Col 13' and '100%'. The right side of the window has scroll bars.

Wait a minute! We removed the module and it didn't give us an error, but then both the area functions are in the session and they have been reported as being in the `AreaFunctions` module. To understand what's going on, we need to understand module autoloading,

PowerShell module autoloading

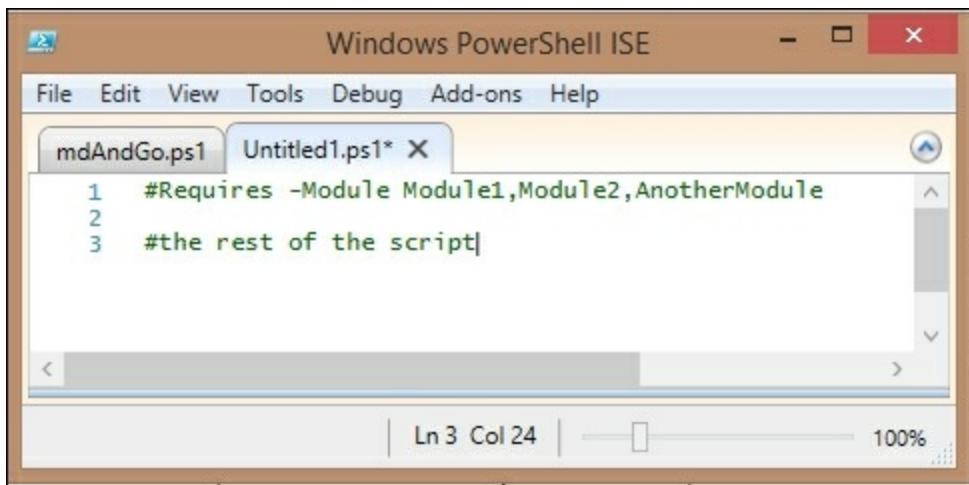
In PowerShell 2.0 on Windows 2008 or Windows 7, there were only a handful of modules installed by the operating system, so knowing what modules were present and what commands were included in each module made sense. In PowerShell 3.0 with the CIM modules, the number of commands available jumped over 2,000, and it became unrealistic for users to be able to remember which of the scores of the modules contained in the commands were needed. To overcome this issue, PowerShell 3.0 introduced the concept of autoloading the modules. In PowerShell 3.0, if you reference a command that is not present in your session, PowerShell will look through the available modules (and in `PSMODULEPATH`), find the module that contains the missing command, and import it silently for you. This means that if your modules are installed in the correct directories, you don't have to use the `Import-Module` statements anymore.

When I first heard about module autoloading, I didn't like it. I thought I wanted to have a control over which modules were loaded and when they were loaded. I still use the `Import-Module` statements in scripts and modules to indicate dependencies, but I often rely on the autoloading behavior.

If you really don't want to use this feature, you can set the `$PSModuleAutoloadingPreference` variable to `None`. This will suppress the autoloading of modules, and force you to explicitly load modules using `Import-Module`.

The #Requires statement

Besides explicitly importing modules into a script, you can annotate the script with a special comment that tells the PowerShell engine that certain modules are required for the script to run. This is done using the `#Requires` statement. As `#Requires` starts with a number sign, it is a comment. However, this special comment has arguments like a cmdlet has. To indicate that a list of modules are required, simply list the module names, as shown in the following screenshot:



A screenshot of the Windows PowerShell ISE application window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. There are two tabs open: "mdAndGo.ps1" and "Untitled1.ps1*". The "Untitled1.ps1*" tab is active and contains the following PowerShell code:

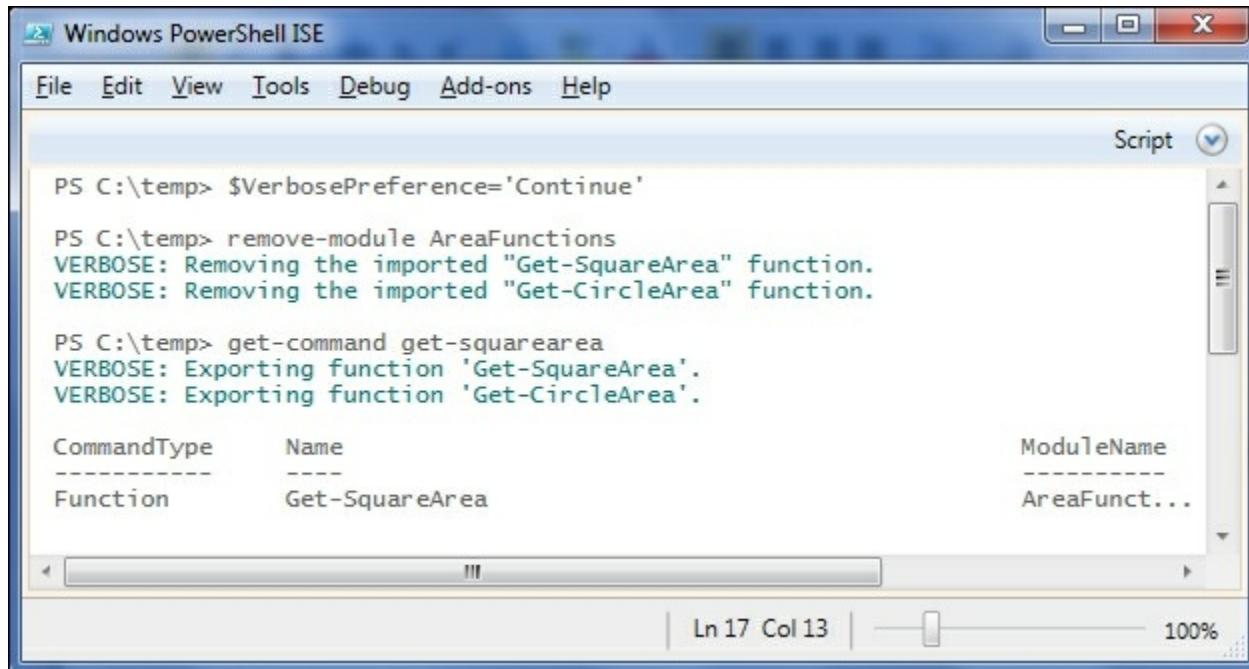
```
1 #Requires -Module Module1,Module2,AnotherModule
2
3 #the rest of the script
```

The status bar at the bottom shows "Ln 3 Col 24" and "100%".

There are several other useful options possible with the `#Requires` statement. To see them, see the `about_Requires` help topic.

Removing a module – take two

We can adjust the `$VerbosePreference`, so that we can see what is going on behind the scenes when we remove and (silently) import a module:



The screenshot shows a Windows PowerShell ISE window. The command `$VerbosePreference='Continue'` is run, followed by `remove-module AreaFunctions`. The output shows verbose messages about removing imported functions: "VERBOSE: Removing the imported 'Get-SquareArea' function." and "VERBOSE: Removing the imported 'Get-CircleArea' function.". Then, the command `get-command get-squarearea` is run, resulting in verbose messages about exporting functions: "VERBOSE: Exporting function 'Get-SquareArea'." and "VERBOSE: Exporting function 'Get-CircleArea'.". Below the command history, a table displays the results of the `get-command` command:

CommandType	Name	ModuleName
Function	Get-SquareArea	AreaFunc...

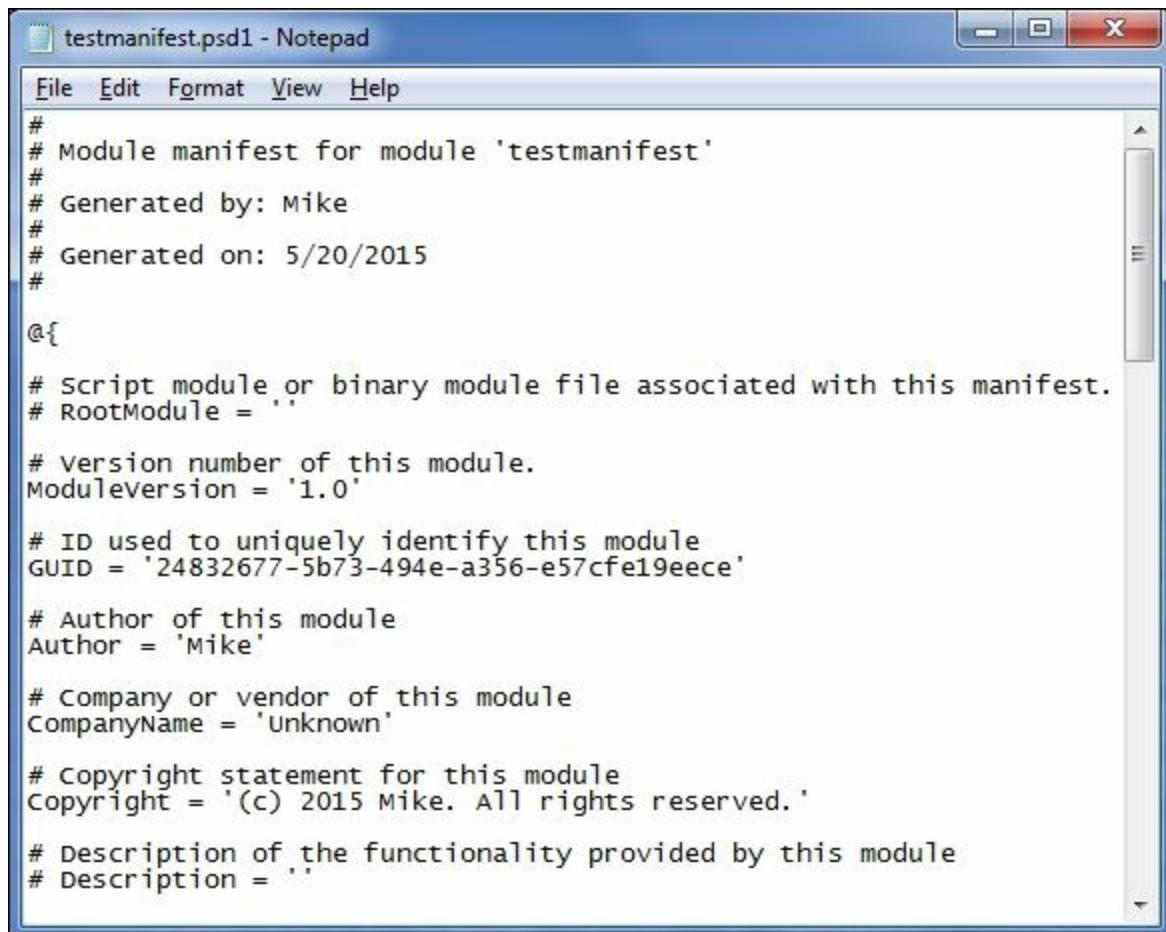
Manifest modules

While script modules are an improvement over dot-sourcing script files, manifest modules take modules to a new level entirely. A module manifest is a configuration file for a module that lets you provide information about the module and gives you control over what happens when the module is loaded. The information you can provide includes things such as the author, company, copyright date, and a GUID identifier for the module. You can specify requirements for the module, such as the minimum PowerShell version or CLR version required. You also can list the functions, aliases, and variables to be exported from your module.

A benefit of using module manifests is that it is much easier for PowerShell to scan a manifest than to parse a script module file. With PowerShell 3.0, and the recent versions of Windows and Windows

Server, the number of delivered modules is very large, and being able to see the contents of a module without parsing hundreds of files is a tremendous time-saver. Manifests also allow you to specify any dependencies for your module, including files that should be executed or loaded when the module is imported.

PowerShell includes a cmdlet called `New-ModuleManifest`, which creates a `.psd1` file for you. Here is the file created by `New-ModuleManifest testmanifest.psd1`:



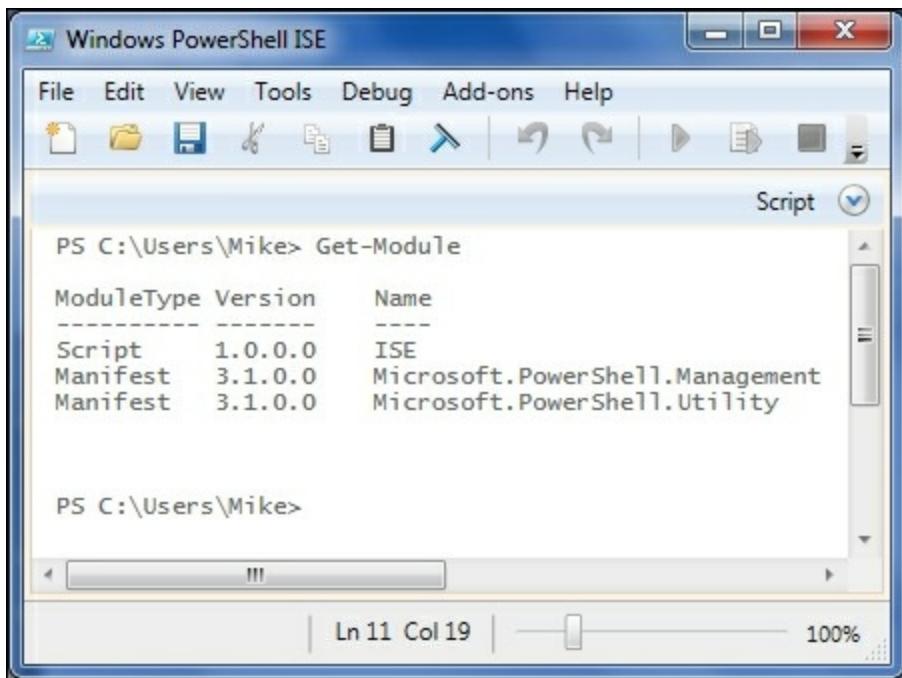
The screenshot shows a Windows Notepad window titled "testmanifest.psd1 - Notepad". The window contains PowerShell module manifest code. The code includes comments for the module's name, author, generation date, and version. It defines variables for the root module file, GUID, author, company name, copyright statement, and a blank description.

```
#  
# Module manifest for module 'testmanifest'  
#  
# Generated by: Mike  
#  
# Generated on: 5/20/2015  
#  
#{@  
  
# Script module or binary module file associated with this manifest.  
# RootModule = ''  
  
# Version number of this module.  
ModuleVersion = '1.0'  
  
# ID used to uniquely identify this module  
GUID = '24832677-5b73-494e-a356-e57cf19eece'  
  
# Author of this module  
Author = 'Mike'  
  
# Company or vendor of this module  
CompanyName = 'Unknown'  
  
# Copyright statement for this module  
Copyright = '(c) 2015 Mike. All rights reserved.'  
  
# Description of the functionality provided by this module  
# Description = ''
```

Listing modules

A module is simply PowerShell's term for a self-contained library or package. PowerShell modules were introduced in Version 2.0. Your operating system is delivered with a number of built-in modules. To see them, you need to use the `Get-Module` cmdlet. If you don't supply any arguments, `Get-Module` outputs a list of all the modules that have been loaded into the current session. To see the list of all the modules that PowerShell can find, you can use the `-ListAvailable` switch.

On my Windows 7 laptop, in a new ISE session, I have only three modules loaded, two "core" modules, and an ISE-specific module, all of which are automatically loaded:



The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The main area is a script editor with the word "Script" and a dropdown arrow. The text pane displays the following command and its output:

```
PS C:\Users\Mike> Get-Module
```

ModuleType	Version	Name
Script	1.0.0.0	ISE
Manifest	3.1.0.0	Microsoft.PowerShell.Management
Manifest	3.1.0.0	Microsoft.PowerShell.Utility

```
PS C:\Users\Mike>
```

At the bottom, there is a status bar showing "Ln 11 Col 19" and "100%".

If we use the `-ListAvailable` switch, we can see that there are several modules in several places that I could import into this session:

Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script

```
PS C:\Users\Mike> Get-Module -ListAvailable

Directory: C:\Users\Mike\Documents\WindowsPowerShell\Modules

ModuleType Version Name                                ExportedCommands
----- -----   ----
Binary      2.0.16.11 ISESteroids                   {Get-PSBreakpoint,...}

Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules

ModuleType Version Name                                ExportedCommands
----- -----   ----
Manifest    1.0.0.0 BitsTransfer                    {Add-BitsFile, Rem...
Manifest    1.0.0.0 CimCmdlets                     {Get-CimAssociated...
Script      1.0.0.0 ISE                           {New-IseSnippet, I...
Manifest    3.0.0.0 Microsoft.PowerShell.Diagnostics {Get-WinEvent, Get...
Manifest    3.0.0.0 Microsoft.PowerShell.Host        {Start-Transcript, ...
Manifest    3.1.0.0 Microsoft.PowerShell.Management {Add-Content, Clea...
Manifest    3.0.0.0 Microsoft.PowerShell.Security     {Get-Acl, Set-Acl, ...
Manifest    3.1.0.0 Microsoft.PowerShell.Utility       {Format-List, Form...
Manifest    3.0.0.0 Microsoft.WSMan.Management     {Disable-WSManCred...
Binary      1.0      PSDesiredStateConfiguration     {Set-DSCLocalConfi...
Script      1.0.0.0 PSDiagnostics                  {Disable-PSTrace, ...
Binary      1.1.0.0 PSScheduledJob                 {New-JobTrigger, A...
Manifest    2.0.0.0 PSWorkflow                     {New-PSWorkflowExe...
Manifest    1.0.0.0 PSWorkflowUtility              {Invoke-AsWorkflow
Manifest    1.0.0.0 TroubleshootingPack            {Get-Troubleshooti...
Manifest    1.0.0.0 WebAdministration             {Start-WebCommitDe...

Directory: C:\Program Files (x86)\Microsoft SQL Server\110\Tools\PowerShell\Modules

ModuleType Version Name                                ExportedCommands
----- -----   ----
Manifest    1.0      SQLASCMDLETS                {Add-RoleMember, B...
Manifest    1.0      SQLPS                      {Backup-SqlDatabase...
```

Ln 56 Col 19 | 100%

The first column of the output shows the type of the module. In the `-ListAvailable` output, we can see the Script, Binary, and Manifest modules. Other types of modules that I don't have on my laptop are CIM and Workflow. These types of modules are described in the following table:

Binary, CIM, and Workflow modules are beyond the scope of Module 1,

please see Module 2 for more on PowerShell modules. As a scripter, you will spend most of your time writing scripts and manifest modules.

Summary

In this chapter, we introduced the concept of modules as containers for multiple functions. We showed how to place the modules so that they can be imported easily and how to control the visibility of items contained in a module.

In the next chapter, we will look at how to interact with files in PowerShell.

For Further Reading

- `Get-Help about_Modules`
- `Get-Help Export-ModuleMember`
- `Get-Help about_Requires`

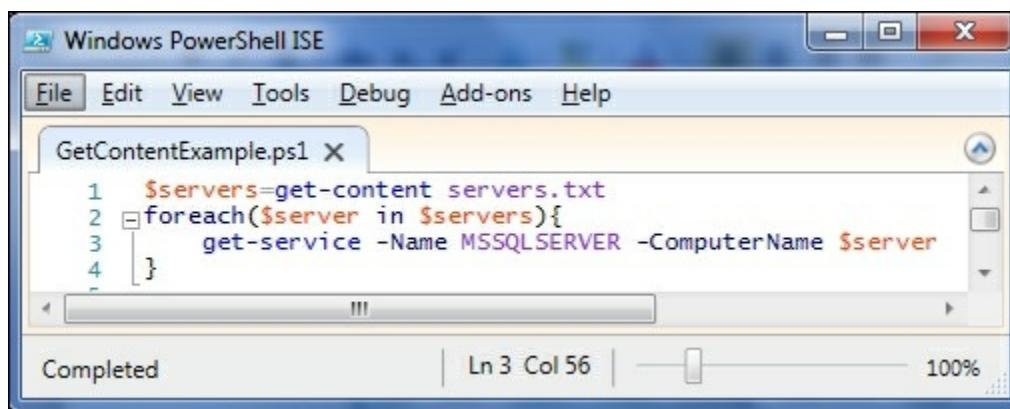
Chapter 9. File I/O

So far, all the input to commands has been in the form of parameters, and the output has been either to the console (with `write-host`), or to the output stream. In this chapter, we will look at some of the ways that PowerShell gives us to work with files. The topics that will be covered include the following:

- Reading and writing text files
- Working with CSV files
- Output redirection
- Reading and writing objects using CLIXML

Reading and writing text files

One method of reading a text file in PowerShell is to use the `Get-Content` cmdlet. `Get-Content` outputs the contents of the file as a collection of strings. If we have a text file called `Servers.txt` that contains the names of the SQL Servers in our environment, we would use the following code to read the file and get the status of the `MSSQLSERVER` service on the servers. Note that with this code, the file will not contain anything but the server names, each on a separate line with no blank lines:



```
Windows PowerShell ISE

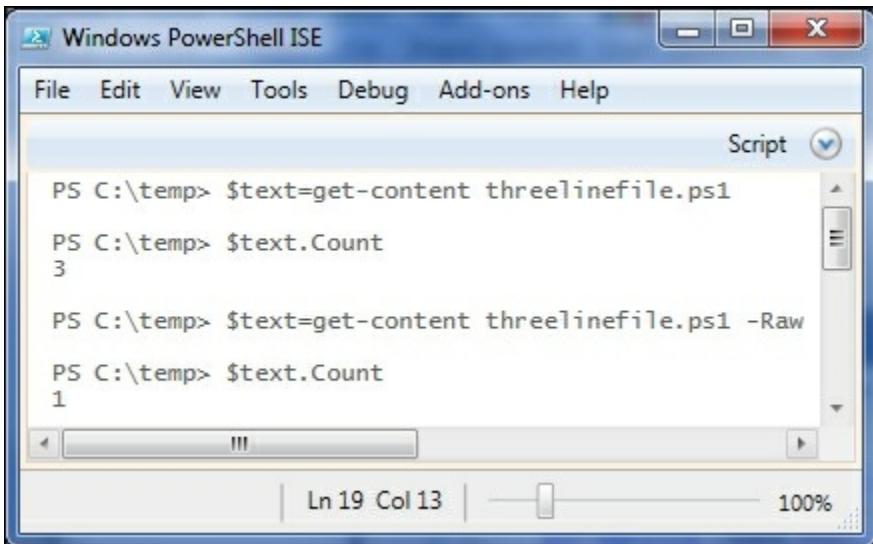
File Edit View Tools Debug Add-ons Help

GetContentExample.ps1 X
1 $servers=get-content servers.txt
2 foreach($server in $servers){
3     get-service -Name MSSQLSERVER -ComputerName $server
4 }
```

Completed | Ln 3 Col 56 | 100%

If you don't want the file to be split into lines, there is a `-Raw` switch parameter that causes the cmdlet to output the entire file as a single

string. As an example, if we have a text file with three lines, we can see the difference between when we use the default operation and when we use `-Raw` by counting the number of strings returned:



```
Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help
Script

PS C:\temp> $text=get-content threelinefile.ps1
PS C:\temp> $text.Count
3

PS C:\temp> $text=get-content threelinefile.ps1 -Raw
PS C:\temp> $text.Count
1

Ln 19 Col 13 | 100%
```

If you want only the beginning or the end of the file, you could use `Select-Object` with the `-First` or `-Last` parameters, but it turns out that `Get-Content` has this covered as well. To get the beginning of the file, you can use the `-TotalCount` parameter to specify how many lines to get. To get the end of the file, use the `-Tail` parameter, instead of the `-TotalCount` parameter, which lets you specify how many of the end lines to output.

Tip

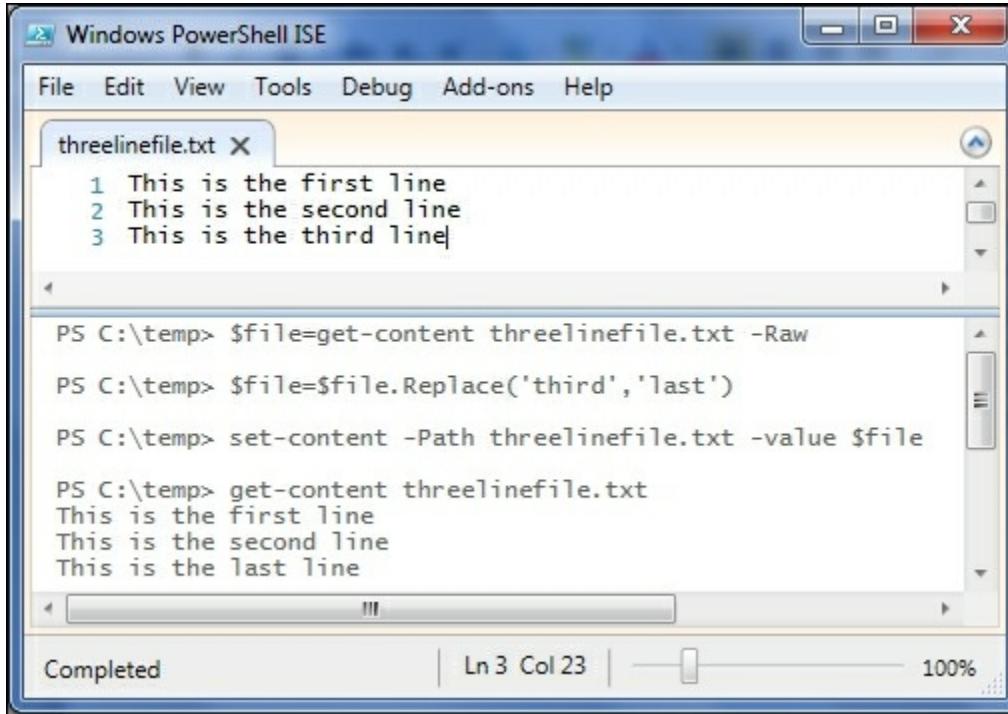
Reminder!

Always filter the pipeline as early as possible. Although, you could achieve the same thing with `Select-Object` as you can with the `-TotalCount` and `-Tail` parameters, filtering at the source (`Get-Content`) will be faster because less objects will have to be created and passed on to the pipeline.

Writing text files

There are several ways to write to a text file. First, if you have the entire

contents of the file in a variable or as the result of a pipeline, you can use the `Set-Content` cmdlet to write the file as a single unit. For instance, you could read a file with `Get-Content`, use the `.Replace()` method to change something, and then write the file back using `Set-Content`:



The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A tab labeled "threelinefile.txt" is open, showing the contents:

```
1 This is the first line
2 This is the second line
3 This is the third line
```

Below the editor, the PowerShell command history and output are displayed:

```
PS C:\temp> $file=get-content threelinefile.txt -Raw
PS C:\temp> $file=$file.Replace('third','last')
PS C:\temp> set-content -Path threelinefile.txt -value $file
PS C:\temp> get-content threelinefile.txt
This is the first line
This is the second line
This is the last line
```

At the bottom, status information is shown: "Completed" and "Ln 3 Col 23".

Another useful feature of `Set-Content` is that you can set the encoding of the file using the `-Encoding` parameter. The possible values for the encoding are:

- ASCII (the default)
- BigEndianUnicode (UTF-16 big-endian)
- Byte
- String
- Unicode (UTF-16 little-endian)
- UTF7
- UTF8
- Unknown

With all these options, you should have no trouble writing in any format that you need.

Working with CSV files

Comma-separated value files, or CSV files, are a mainstay of the PowerShell world. In the next two sections, we will see how they are very useful, both as input and as output.

Output to CSV for quick reports

If your workplace is anything like mine, you probably work with people who want reports about what is going on. Writing "real" reports in SQL Server reporting services is an option if your data is accessible to SQL server, but they take a while to write and deploy. Obviously, there are reporting packages that you can use as well, such as Crystal Reports, but they can be expensive and take time to write and deploy a report. Most people in IT, though, are fine with Excel as a document format and can work with the data in Excel to create a report.

Outputting objects to CSV files in PowerShell is very simple. The **Export-Csv** cmdlet looks at the properties of the first object in the pipeline and creates a CSV file with a header row containing the names of these properties. It writes the values of these properties in successive lines of the file. Since it uses all the properties of the (first) object, you will probably want to "narrow" down the object using `Select-Object` and the `-Property` parameter to limit the properties that show up in your CSV file.

For example, if you wanted to create a file with the names and lengths of the files in a folder, you could use the following code:

The screenshot shows the Windows PowerShell Integrated Scripting Environment (ISE). At the top, the title bar reads "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The main area has two panes. The left pane displays the script "ExportCSVExample.ps1" with the following content:

```
1 dir c:\temp | select-object -Property Name,Length |
2     Export-csv C:\temp\fileList.csv
```

The right pane shows the output of the script:

```
PS C:\Users\Mike> get-content c:\temp\filelist.csv -TotalCount 5
#TYPE Selected.System.IO.DirectoryInfo
"Name","Length"
"abcde.txt","0"
"adv_event1.ps1","2399"
"AreaFunctions.psm1","279"
```

At the bottom of the ISE window, status bars indicate "Completed", "Ln 8 Col 19", and "100%".

You can see from the output of the `Get-Content` cmdlet that there is a line at the top that indicates the type of objects that were output. You can suppress this line by including the `-NoTypeInformation` switch. You can also see that the column headings and the values are enclosed in quotes and separated by commas. It's possible to change the delimiter to something other than a comma using the `-Delimiter` parameter, but you don't have the option to suppress the quotes.

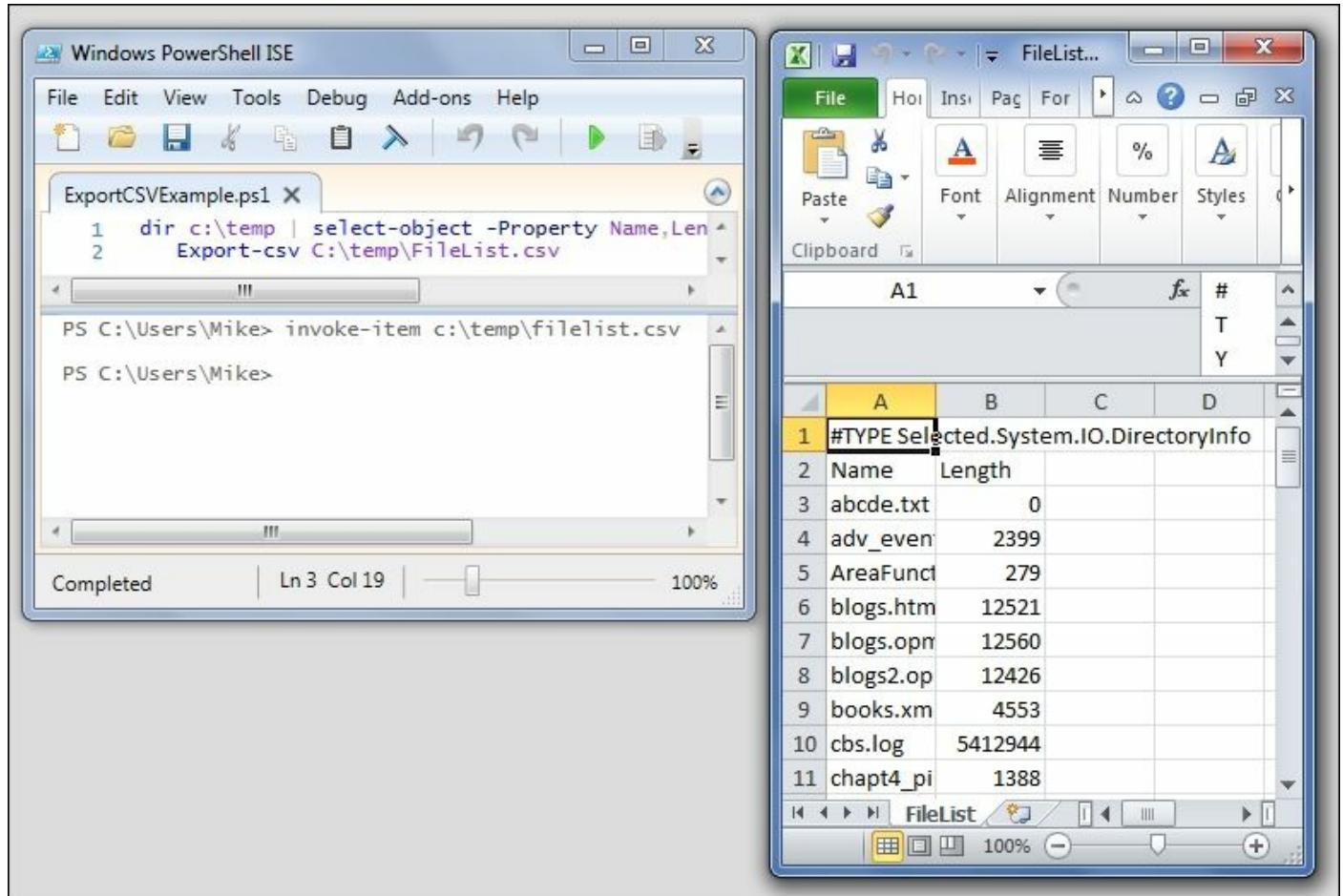
Tip

Although the cmdlets refer to CSV, using a delimiter such as `t (a tab) would obviously create a tab-separated value file or a TSV. For some kinds of data, TSV might be a preferred format, and programs like Excel are able to load TSV files just fine.

The Invoke-Item cmdlet

You can easily open a CSV file in Excel (or whatever application is associated with CSV files on your system) using the `Invoke-Item` cmdlet. `Invoke-Item` performs the default action associated with a particular file. You can think of it as double-clicking on an item in the

File Explorer. If you have Excel, the default action for CSV files is to open them in Excel. To easily open the output file we created in the last section, we would use `Invoke-Item c:\temp\filelist.csv`, and it would pop up looking like a spreadsheet as follows:



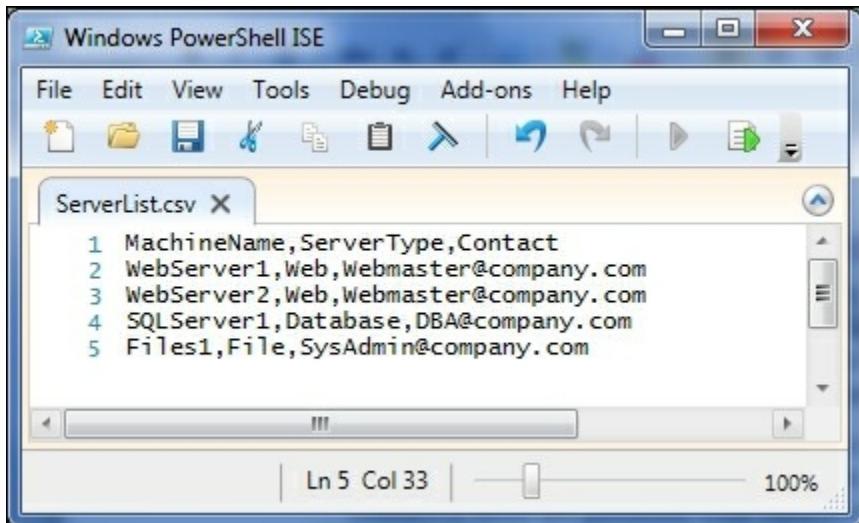
At this point, it can be manipulated just like a normal spreadsheet, so it's easy for people who haven't learned PowerShell yet to work with it.

Import from CSV for quick objects

Since PowerShell cmdlets work on objects, it's convenient to be able to read a file as a sequence of objects rather than as strings, like you would from `Get-Content`. Just as `Export-CSV` takes objects and writes them to a file, `Import-CSV` reads a CSV file and outputs objects with the properties that are named in the header.

An easy example to understand would be a CSV file that contains a

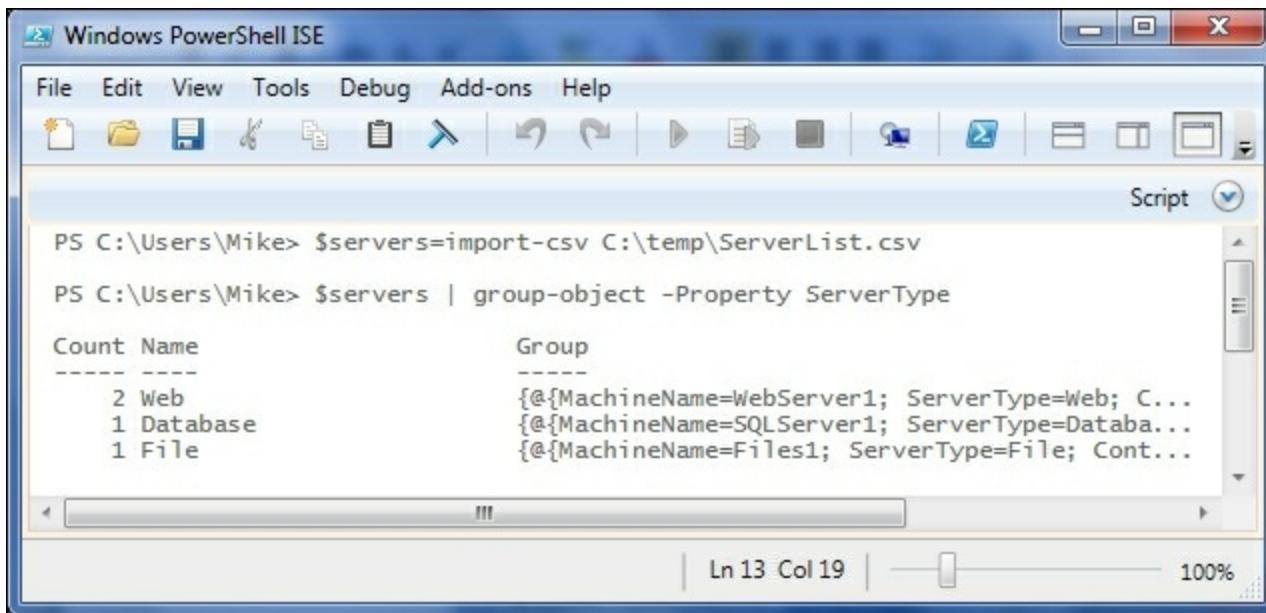
server list with the name of each server, as well as the type of the server (Web, Database, or File), and whom to contact, if there are problems with each server:



The screenshot shows the Windows PowerShell Integrated Scripting Environment (ISE) window titled "Windows PowerShell ISE". A file named "ServerList.csv" is open in the editor. The content of the CSV file is as follows:

	MachineName	ServerType	Contact
1	WebServer1	Web	Webmaster@company.com
2	WebServer2	Web	Webmaster@company.com
3	SQLServer1	Database	DBA@company.com
4	Files1	File	SysAdmin@company.com

With this CSV file, we can easily read this into a variable using `Import-Csv`. As `Import-Csv` creates objects, we can refer to the rows of this CSV file using properties. Getting a summary of our servers using the `Group-Object` cmdlet is a simple matter:



The screenshot shows the Windows PowerShell ISE window titled "Windows PowerShell ISE". The session pane contains the following PowerShell commands:

```
PS C:\Users\Mike> $servers=import-csv C:\temp\ServerList.csv
PS C:\Users\Mike> $servers | group-object -Property ServerType
```

After running these commands, the results are displayed in the host pane:

Count	Name	Group
2	Web	{@{MachineName=WebServer1; ServerType=Web; C...}
1	Database	{@{MachineName=SQLServer1; ServerType=Database...}
1	File	{@{MachineName=Files1; ServerType=File; Cont...}}

PowerShell streams and redirection

We've talked about the output stream, but it turns out that PowerShell has quite a few streams besides this. Here's the list, including one that was introduced in PowerShell 5:

Stream	Number	Contents
Output	1	Output from commands
Error	2	Error messages
Warning	3	Warning messages
Debug	4	Debug messages
Verbose	5	Verbose output
Information	6	General information (PowerShell 5)

Similar to DOS, you can redirect streams to a file using the greater-than symbol (>). For example, you can send a directory listing to a file as follows:

```
Dir c:\temp >c:\temp\files.txt
```

The output is formatted according to standard PowerShell formatting, but since it's just a text file, you can't recreate the objects that you started with. Also, unlike with `Set-Content`, you can't control the output encoding, which is always Unicode if you use the redirection operator.

Other types of redirection operators

In addition to the single greater-than symbol, there are a number of other

redirection operators that append (using two greater-than symbols) and combine streams (such as `2>&1`, which merges the error stream to the output stream using the numbers found in the table). As there are a number of streams and a variety of operators, you should refer to the `about_redirection` help topic for a complete description.

The `out-file` cmdlet

A final way to redirect output is to use the `out-file` cmdlet. Unlike the `redirect` operator, `out-file` includes an `-Encoding` parameter that lets you change the encoding. The values for this parameter are the same as the `Set-Content` cmdlet. The main difference between `Set-Content` and `Out-File` is that `Out-File` accepts pipeline input. So, you will use `Out-File` as the last command in a pipeline to send the output to a file.

For instance, the redirection we performed with the redirection operator would be expressed using `Out-File` as follows:

```
Dir c:\temp | out-file C:\temp\files.txt
```

I don't know about you, but this is a lot easier for me to read. If you want to append to an existing file, the `-Append` switch is what you're looking for. You don't get to combine streams as you can with the redirection operators, but for general purpose output redirection, `Out-File` is a good candidate.

CLIXML – a special type of XML

You've seen how we can use `Export-Csv` to write objects to a file and that we can read these objects back using `Import-Csv`. One limitation of using the CSV format is that it is a flat format, meaning that you get a list of properties that have simple values. If you need to output arbitrary objects to a file and need to store more interesting values than strings, you should probably look at the ClixML cmdlets, `Export-CLIXML` and `Import-CLIXML`. ClixML is an XML-based format to serialize objects.

When you export objects using `Export-CLIXML`, it looks at the list of properties that may be value properties (such as `string` or `int` values) or they may be objects themselves. If they are objects, `Export-CLIXML` will look at the properties of the properties. It will continue this process until it reaches a predetermined depth. The default value of depth is "2", but you can override this using the `-Depth` parameter.

To see how this works, let's get a particular service and export it to a file using this command:

```
get-service MSSQLSERVER | Export-CLIXML -path  
C:\temp\SQL.clixml
```

Looking at the following output, you can see that it's XML, but it's still fairly readable. The file includes the list of type names (the type of the object as well as any ancestor types), followed by a list of properties. Some of them start with an `S` or a `B`, denoting String or Boolean, and others start with `OBJ` or `NIL` meaning objects that are either populated or empty:

The screenshot shows the Windows PowerShell Integrated Scripting Environment (ISE) window. The title bar reads "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations like Open, Save, and Print. The main code editor pane displays the XML content of a CLIXML file named "SQL.clixml". The XML structure represents two objects, likely ServiceController instances, with their properties and nested types. The status bar at the bottom shows "Completed" on the left, "Ln 1 Col 1" in the center, and "100%" on the right.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <k0:objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">
3   <k0:Obj RefId="0">
4     <TN RefId="0">
5       <T>System.ServiceProcess.ServiceController</T>
6       <T>System.ComponentModel.Component</T>
7       <T>System.MarshalByRefObject</T>
8       <T>System.Object</T>
9     </TN>
10    <ToString>MSSQLSERVER</ToString>
11    <Props>
12      <B N="CanPauseAndContinue">false</B>
13      <B N="CanShutdown">false</B>
14      <B N="CanStop">false</B>
15      <S N="DisplayName">SQL Server (MSSQLSERVER)</S>
16      <Obj N="DependentServices" RefId="1">
17        <TN RefId="1">
18          <T>System.ServiceProcess.ServiceController[]</T>
19          <T>System.Array</T>
20          <T>System.Object</T>
21        </TN>
22        <LST>
23          <Obj RefId="2">
24            <TNRef RefId="0" />
25            <ToString>SQLSERVERAGENT</ToString>
26            <Props>
27              <B N="CanPauseAndContinue">false</B>
28              <B N="CanShutdown">false</B>
              ...</Props>
            </Obj>
          </LST>
        </Obj>
      </Props>
    </Obj>
  </Props>
</Obj>
</k0:objs>
```

Reading a CLIXML file with `Import-CLIXML` builds a complex object, but it's not quite the same object that you started with for three reasons, as follows:

- The object was only written to a limited depth, so we could have lost some details
- The methods are not included in the recreated object, only the properties
- Since we didn't use the constructor for the type of object we need, the type of the object is now slightly different

We can see the differences using `get-member` on the imported object:

The screenshot shows a Windows PowerShell ISE window. The command \$Sql=Import-Clixml C:\temp\SQL.clixml is run, followed by \$sql | get-member. The output shows the members of the Deserialized.System.ServiceProcess.ServiceController type.

Name	MemberType	Definition
GetType	Method	type GetType()
ToString	Method	string ToString(), string ToString(string form...
Name	NoteProperty	System.String Name=MSSQLSERVER
RequiredServices	NoteProperty	Deserialized.System.ServiceProcess.ServiceCont...
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.ServiceCont...
DisplayName	Property	System.String {get;set;}
MachineName	Property	System.String {get;set;}
ServiceName	Property	System.String {get;set;}
ServicesDependedOn	Property	Deserialized.System.ServiceProcess.ServiceCont...
ServiceType	Property	Deserialized.System.ServiceProcess.ServiceType...
Site	Property	{get;set;}
Status	Property	Deserialized.System.ServiceProcess.ServiceCont...

Tip

You try it!

Use the `Get-Member` cmdlet along with the `Get-Service` cmdlet to see what members are present in an actual `ServiceController` object. Then, compare this with the previous screenshot to see that the `Deserialized` object is actually different.

Note that the type of the object now starts with **Deserialized**. This is because the process of writing an object is called serialization, and the process of recreating an object from a serialized format is called deserialization. You might, also, hear such objects being referred to as "rehydrated" or "unwired". The point is that this object, for instance, is

no longer connected to the MSSQLSERVER service on my laptop like the original object was.

Summary

PowerShell has a very rich set of cmdlets that deal with file input and output. We've seen how to read and write text files, CSV files, and CLIXML files. We, also, looked at how to use redirection to cause PowerShell to write streams to files or combine streams together.

In the next chapter, we will look at one way by which PowerShell can communicate with different kinds of software and components on a computer, using WMI and CIM. These capabilities greatly expand PowerShell's power, and causes it to be an almost universal tool in the Microsoft environment.

For further reading

- Get-Help Get-Content
- Get-Help Set-Content
- Get-Help Out-File
- Get-Help about_Redirection
- Get-Help Export-CSV
- Get-Help Import-CSV
- Get-Help Export-CLIXML
- Get-Help Import-CLIXML

Chapter 10. WMI and CIM

We've looked at several kinds of cmdlets so far in Module 1, but until now, each cmdlet has been working with a specific kind of object. In this chapter, we will look at the WMI and CIM cmdlets, which will allow us to look at a seemingly unlimited range of object types. In this chapter, we will cover the following topics:

- What are WMI and CIM?
- A WMI/CIM glossary
- Retrieving objects via WMI and CIM
- Calling methods via WMI and CIM
- The CDXML modules

What is WMI?

Getting objects from cmdlets is what PowerShell is all about. Unfortunately, there are more kinds of objects that we might be interested in than PowerShell has cmdlets for. This was definitely the case during the time of PowerShell 1.0, which only had less than a hundred cmdlets. What do we do about the objects for which PowerShell doesn't have an explicit cmdlet? The answer involves **Windows Management Instrumentation (WMI)**. WMI was introduced as a common interface to manage operating system objects all the way back in Windows NT. WMI provides a uniform way to retrieve objects, which allows us to inspect and even change the state of components, processes, and other objects. Before we look at specific WMI information, it's important to understand how objects are stored in WMI.

WMI organization

The WMI objects are stored in a repository that is local to a computer system. The repository is divided into several namespaces, each of which provides different objects. Inside a namespace there are the WMI classes, each of which describes a certain type of object. Finally, there

are the WMI instances, which are specific examples of a WMI class.

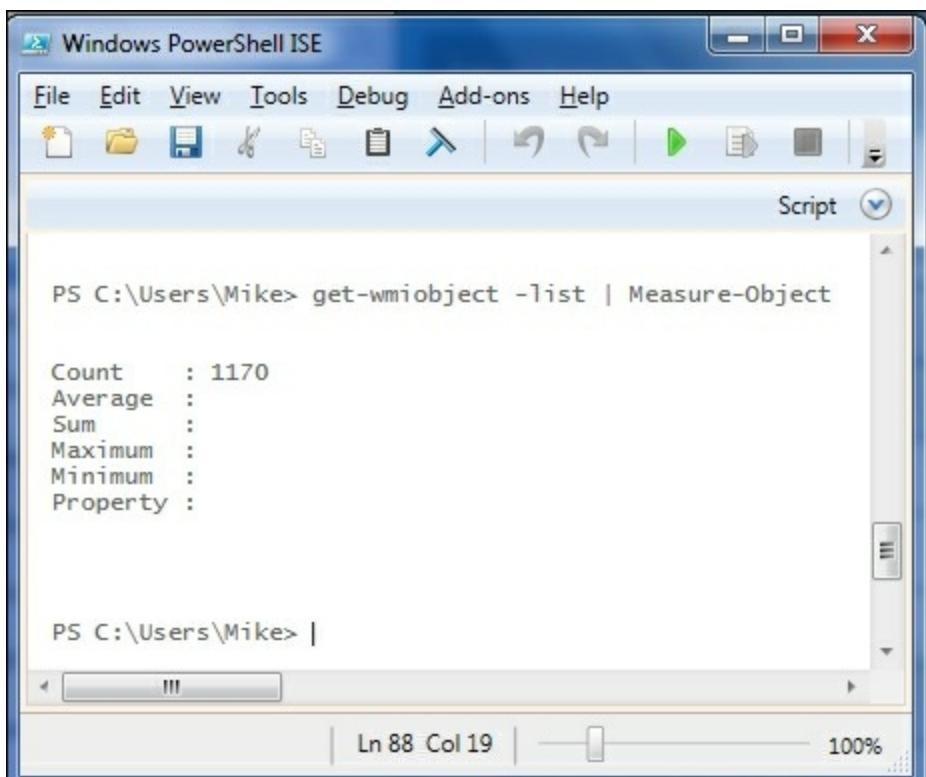
An analogy would help to make more sense of this. If we think of the WMI repository like a filing cabinet, the comparison might go something as follows:

- The repository is a filing cabinet
- A namespace is a drawer in the cabinet
- A class is a folder in the drawer
- An instance is a piece of paper in the drawer

The information that you want is found on the piece of paper, that is, in the WMI instance.

Finding WMI classes

The first thing to know about the WMI classes is that there are a ton of them. So many, in fact, that you will almost certainly never need to use most of them. As there are so many classes, finding the class that you want to work with can be interesting. The `Get-WMIObject` cmdlet has a parameter called `-List`, which lets you get a list of the classes installed on your computer. Combining this cmdlet with `Measure-Object` shows that my computer (Windows 7) has over a thousand classes in the default namespace:



A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The main window shows a command prompt at PS C:\Users\Mike>. The command entered is `get-wmiobject -list | Measure-Object`. The output shows statistical information for the 1170 WMI classes found:

```
PS C:\Users\Mike> get-wmiobject -list | Measure-Object

Count      : 1170
Average    :
Sum        :
Maximum   :
Minimum   :
Property  :

PS C:\Users\Mike> |
```

The status bar at the bottom indicates "Ln 88 Col 19" and "100%".

You clearly don't want to look through a list of so many items, so you can narrow down the list with the `-Class` parameter, which accepts wildcards. For instance, to find classes that deal with the processor, you might do this:

The screenshot shows a Windows PowerShell ISE window. The title bar reads "Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". The toolbar contains various icons for file operations like Open, Save, Print, and Help. The main pane displays a command prompt and its output:

```
PS C:\Users\Mike> get-wmiobject -list -Class *processor*
```

NameSpace: ROOT\cimv2

Name	Methods	Properties
CIM_Processor	{SetPowerState, R...	{AddressWidth, Avai...
Win32_Processor	{SetPowerState, R...	{AddressWidth, Arch...
Win32_ComputerSystemProcessor	{}	{GroupComponent, Pa...
CIM_AssociatedProcessorMemory	{}	{Antecedent, BusSpe...
Win32_AssociatedProcessorMemory	{}	{Antecedent, BusSpe...
Win32_PerfFormattedData_Counters...	{}	{BuildScatterGather...
Win32_PerfRawData_Counters_PerPr...	{}	{BuildScatterGather...
Win32_PerfFormattedData_Counters...	{}	{BuildScatterGather...
Win32_PerfRawData_Counters_PerPr...	{}	{BuildScatterGather...
Win32_PerfFormattedData_Counters...	{}	{C1TransitionsPerse...
Win32_PerfRawData_Counters_Proce...	{}	{C1TransitionsPerse...
Win32_PerfFormattedData_PerfOS_P...	{}	{C1TransitionsPerse...
Win32_PerfRawData_PerfOS_Processor	{}	{C1TransitionsPerse...

The first two in the list (`CIM_Processor` and `Win32_Processor`) are the results that are most useful in this case. Depending on what wildcard you use, you may need to look at several classes to find the one that is most appropriate.

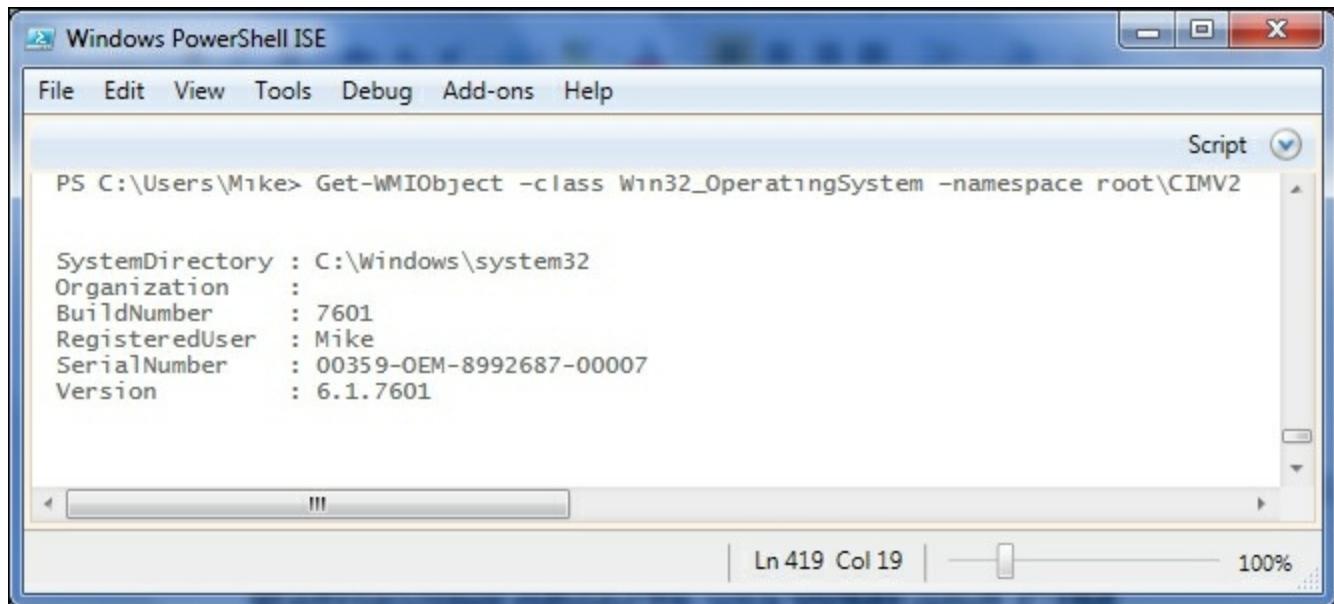
Tip

The classes beginning with Win32 are based on the classes beginning with CIM. Generally, I stick with the Win32 classes, but they usually have almost the same information.

Retrieving objects with Get-WMIOBJECT

If you know the name of a class and what namespace that class belongs to, it is simple to retrieve the class using the `Get-WMIOBJECT` cmdlet. For instance, there is a class that describes the installed operating system called `Win32_OperatingSystem` contained in the default (`root\CIMV2`) namespace. To retrieve this class, either of these command-lines will work:

```
Get-WMIOBJECT -class Win32_OperatingSystem  
Get-WMIOBJECT -class Win32_OperatingSystem -namespace  
root\CIMV2
```



The screenshot shows a Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with icons for New, Open, Save, and Run is visible. The main area has tabs for "Script" and "Output". The "Script" tab contains the PowerShell command: "PS C:\Users\Mike> Get-WMIOBJECT -class Win32_OperatingSystem -namespace root\CIMV2". The "Output" tab displays the results of the command, which are the properties of the Win32_OperatingSystem class. The properties listed are:

Property	Value
SystemDirectory	C:\Windows\system32
Organization	:
BuildNumber	7601
RegisteredUser	Mike
SerialNumber	00359-OEM-8992687-00007
Version	6.1.7601

The output here is formatted, so you will probably want to use `Select-Object -property *` to see all the properties. In this case, there are 73 properties, which are too many for a screenshot.

Tip

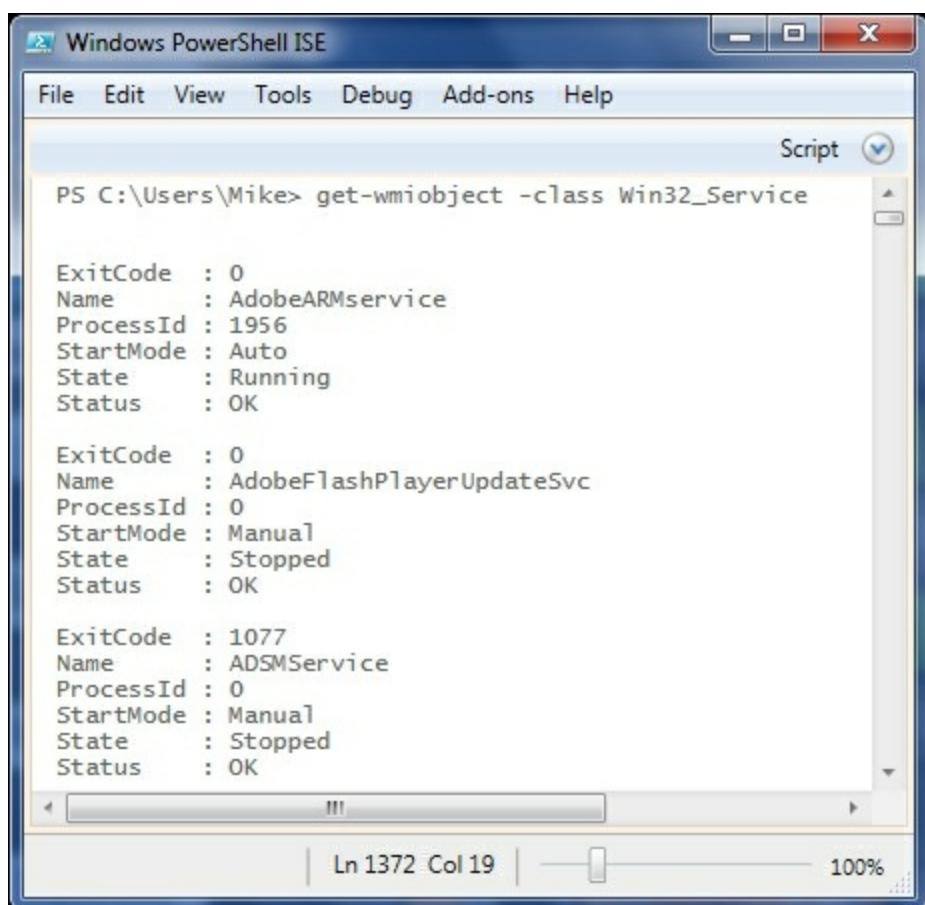
You try it!

Use `Get-WMIOBJECT` with the `Win32_OperatingSystem` class and look

through the properties. You might make a note of the properties that you think would be useful. As you use WMI (and CIM) more and more, you will find a number of classes and properties that you will come back to over and over again. Hence, having a list of "favorites" can be a real life-saver.

One thing you probably noticed with `Win32_OperatingSystem` is that only one object (instance) was retrieved by the cmdlet. This makes sense, because there's only one operating system running on a computer at a time (not counting VMs, of course). The other classes might return zero, one, or any number of instances.

A class that should be easy to understand is the `Win32_Service` class. This class represents the Windows services that are installed in your operating system. You can see the beginning of the formatted output in the following screenshot:



The screenshot shows a Windows PowerShell ISE window with the title bar "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu next to the menu bar is set to "Script". The main pane displays the command "PS C:\Users\Mike> get-wmiobject -class Win32_Service" followed by its output. The output lists three service objects with their properties:

```
ExitCode : 0
Name      : AdobeARMservice
ProcessId : 1956
StartMode  : Auto
State     : Running
Status    : OK

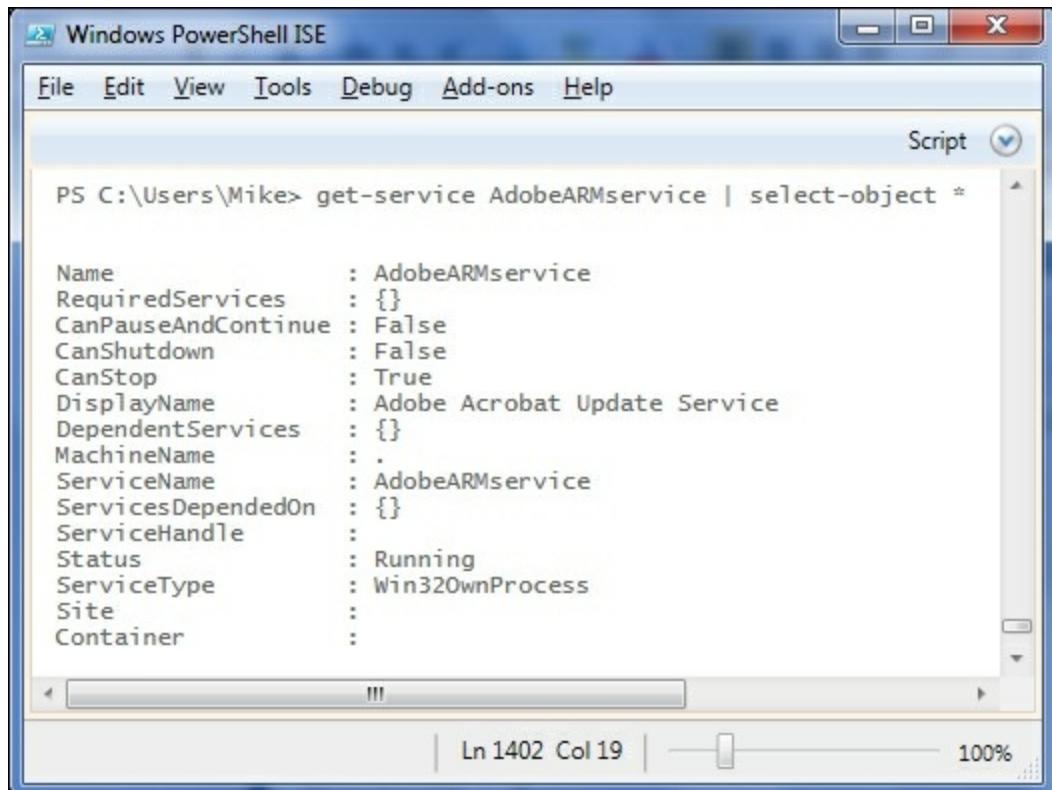
ExitCode : 0
Name      : AdobeFlashPlayerUpdateSvc
ProcessId : 0
StartMode  : Manual
State     : Stopped
Status    : OK

ExitCode : 1077
Name     : ADSMService
ProcessId : 0
StartMode : Manual
State    : Stopped
Status   : OK
```

At the bottom of the window, status bars show "Ln 1372 Col 19" and "100%".

You might ask why we would use WMI to get information about

services when we have a perfectly good cmdlet (`Get-Service`) that is designed to do this. Even though we can see only a few properties of each WMI service object in the preceding output, there's enough information to illustrate an important point. The first object in the output, `AdobeARMService`, shows six properties. Comparing these properties with the full output of the `Get-Service` cmdlet reveals that the `Name` and `State` properties of the WMI object have analogous properties (`Name` and `Status`) in the `Get-Service` object, but the other four properties (including `ProcessID`) are missing from `Get-Service`:



The screenshot shows a Windows PowerShell ISE window. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar button labeled "Script" is visible. The main pane displays the following PowerShell command and its output:

```
PS C:\Users\Mike> get-service AdobeARMService | select-object *
```

Name	Value
Name	: AdobeARMService
RequiredServices	: {}
CanPauseAndContinue	: False
CanShutdown	: False
CanStop	: True
DisplayName	: Adobe Acrobat Update Service
DependentServices	: {}
MachineName	: .
ServiceName	: AdobeARMService
ServicesDependedOn	: {}
ServiceHandle	: .
Status	: Running
ServiceType	: Win32OwnProcess
Site	: .
Container	: .

The status bar at the bottom indicates Ln 1402 Col 19 and 100%.

Remember that WMI is a very mature technology that has been in use for over 20 years. As WMI is so well established, most aspects of the Windows ecosystem is covered very well by WMI.

Tip

Tip!

In addition to keeping track of the interesting WMI classes, you might also want to make a note of the properties of WMI instances that are

missing from the objects output by other PowerShell cmdlets. Often, WMI provides the easiest way to get some information.

Getting the right instance

With `Win32_ComputerSystem`, we didn't have to worry about which instance was returned because there was only one. The next class we looked at was `Win32_Service`, which returned multiple objects, one for each service on the computer. In order to select specific instances, we can use the `-Filter` parameter.

Since WMI covers thousands of different kinds of classes, it would be impossible for `Get-WMIOBJECT` to have specific parameters to help us narrow down which objects we would like to see. For instance, if we use `Get-Service` to look at services, we can use the `-Name` parameter to filter by name. Not all WMI objects have a name property, so it wouldn't make sense to have a parameter to filter by this single property. The solution is to have a query language called **WMI Query Language (WQL)**, similar to SQL, and a single parameter called `-Filter` that takes the WHERE clause of the WQL query in order to filter the objects.

WQL syntax

WQL syntax is very simple to use, and is very similar to SQL. In the WQL filter, you can use the property names of classes, constants, and operators to build the conditions you are looking for. For example, to return only the `AdobeARMService` instance from the `Win32_Service` class, you could use a filter of `"Name='AdobeARMService'"`. Note that I have used double quotes around the filter and single quotes around the string value of the `Name` property. The full `Get-WMIOBJECT` statement and output looks as follows:

A screenshot of the Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu next to the menu bar says "Script". The main pane displays the following PowerShell command and its output:

```
PS C:\Users\Mike> get-wmiobject -class Win32_Service -Filter "Name='AdobeARMService'"  
ExitCode : 0  
Name      : AdobeARMService  
ProcessId : 1956  
StartMode  : Auto  
State     : Running  
Status    : OK
```

The status bar at the bottom shows "Ln 1417 Col 19" and "100%".

Filters can be more complex as well. Consider the `Win32_LogicalDisk` class, which contains an instance for each drive on the local computer:

A screenshot of the Windows PowerShell ISE window. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu next to the menu bar says "Script". The main pane displays the following PowerShell command and its output:

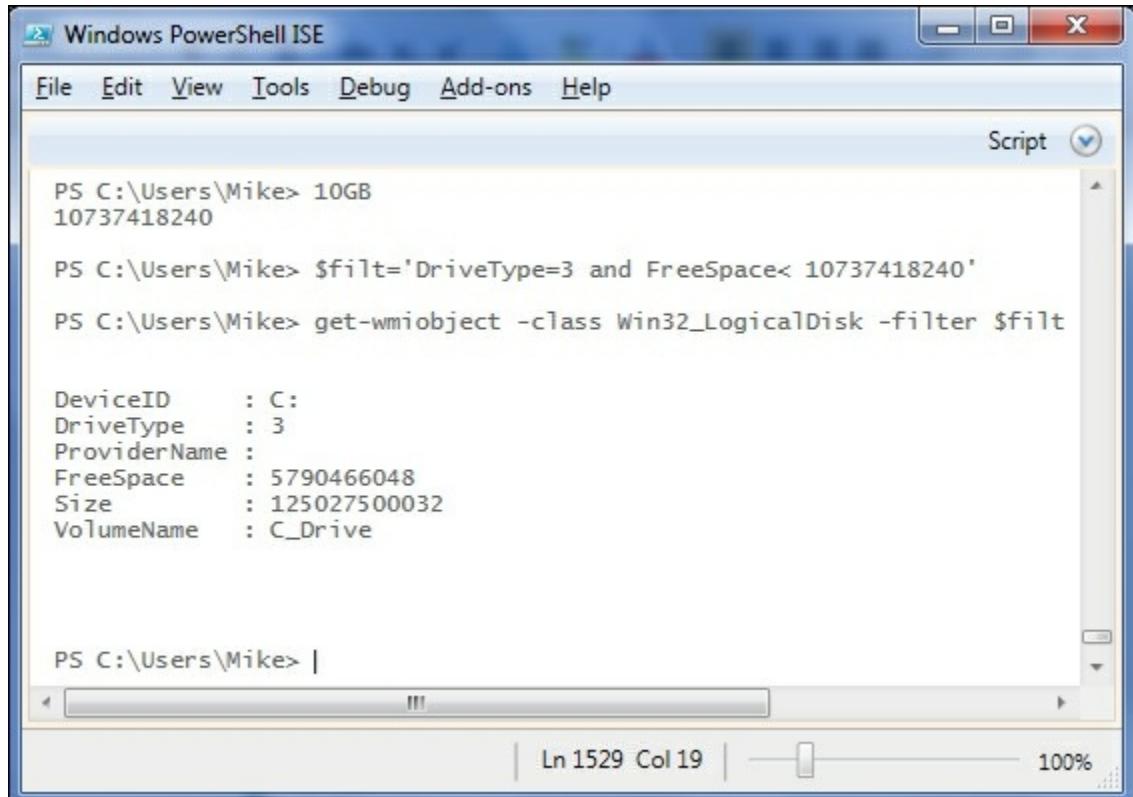
```
PS C:\Users\Mike> get-wmiobject -class Win32_LogicalDisk
```

DeviceID	DriveType	ProviderName	FreeSpace	Size	VolumeName
C:	3		5793271808	125027500032	C_Drive
D:	3		161794637824	354107752448	Data
E:	5				

The status bar at the bottom shows "Ln 1511 Col 19" and "100%".

If we wanted to list all the drives that had less than 10GB free and didn't care about the removable, we could use a filter as follows:

```
-Filter "DriveType=3 and FreeSpace< 10737418240"
```



The screenshot shows a Windows PowerShell ISE window. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu labeled 'Script' is open. The command history shows:

```
PS C:\Users\Mike> 10GB  
10737418240  
PS C:\Users\Mike> $filt='DriveType=3 and FreeSpace< 10737418240'  
PS C:\Users\Mike> get-wmiobject -class Win32_LogicalDisk -filter $filt  
  
DeviceID      : C:  
DriveType     : 3  
ProviderName  :  
FreeSpace     : 5790466048  
Size          : 125027500032  
VolumeName    : C_Drive
```

The status bar at the bottom indicates Ln 1529 Col 19 and a zoom level of 100%.

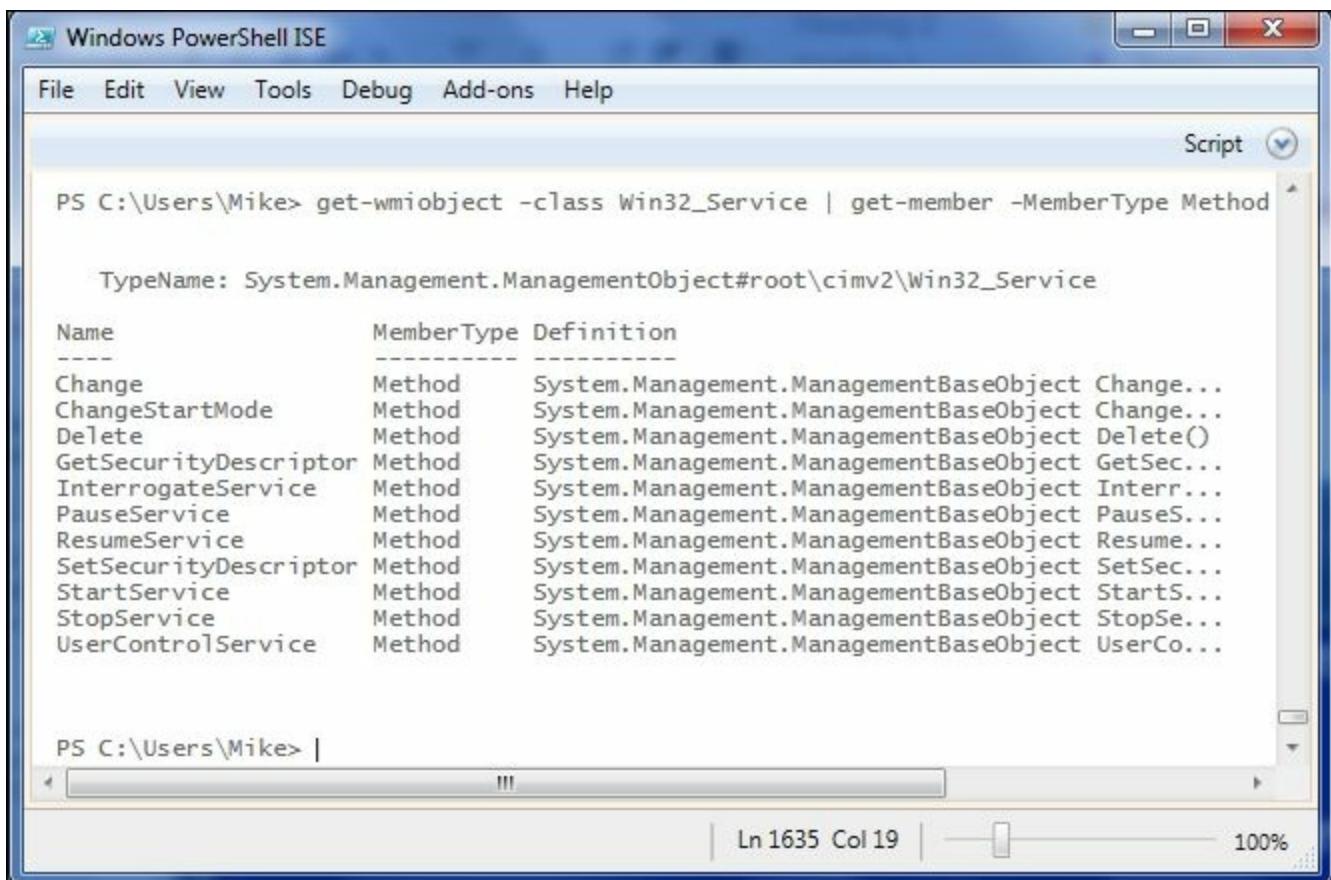
Here, you can see that I have calculated the value of 10GB because WQL doesn't understand units, and also, I have used a variable to hold the filter in order to keep the line shorter.

Tip

Although WQL filters look like SQL syntax, there are crucial differences. You can use comparison operators (=, <>, >, <, <=, >=, LIKE) in the expressions and can link multiple expressions together with AND and OR. You can also group expressions using parentheses. Another important thing to remember is that the WQL strings use backslash as the escape character. For more details, including how to use the WQL queries, refer to the [About_WQL](#) help topic.

Calling methods

When dealing with WMI, there are two kinds of methods that come into play, instance methods and class methods. Calling the WMI instance methods is similar to calling methods on any PowerShell object, that is, using dot-notation. For instance, WMI objects from the `Win32_Service` class have a `StartService()` method defined in them, which we can verify using `Get-Member`:



The screenshot shows a Windows PowerShell ISE window. The command entered is `get-wmiobject -class Win32_Service | get-member -MemberType Method`. The output displays the methods available for the `Win32_Service` class, including `Change`, `ChangeStartMode`, `Delete`, `GetSecurityDescriptor`, `InterrogateService`, `PauseService`, `ResumeService`, `SetSecurityDescriptor`, `StartService`, `StopService`, and `UserControlService`. Each method is listed with its definition as a `System.Management.ManagementBaseObject`.

```
PS C:\Users\Mike> get-wmiobject -class Win32_Service | get-member -MemberType Method
TypeName: System.Management.ManagementObject#root\cimv2\Win32_Service

Name           MemberType  Definition
----           Method     System.Management.ManagementBaseObject Change...
Change         Method     System.Management.ManagementBaseObject Change...
ChangeStartMode Method    System.Management.ManagementBaseObject Delete()
Delete         Method    System.Management.ManagementBaseObject GetSec...
GetSecurityDescriptor Method  System.Management.ManagementBaseObject Interr...
InterrogateService Method  System.Management.ManagementBaseObject PauseS...
PauseService   Method    System.Management.ManagementBaseObject Resume...
ResumeService  Method    System.Management.ManagementBaseObject SetSec...
SetSecurityDescriptor Method System.Management.ManagementBaseObject StartS...
StartService   Method    System.Management.ManagementBaseObject StopSe...
StopService   Method    System.Management.ManagementBaseObject UserCo...
```

To call this method, we can store the instance in a variable and use the variable to call the method. Note that the method outputs a structured result type. The `ReturnValue` property of the output gives us the outcome of the operation. A value of zero means success. An example of a successful method invocation can be seen in the following screenshot:

The screenshot shows a Windows PowerShell ISE window titled "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations. The main pane displays the following PowerShell session:

```
PS C:\Windows\system32> $svc=get-wmiobject -class win32_Service -filter "Name='MSSQLSERVER'"  
PS C:\Windows\system32> $svc.StartService()  
  
__GENUS          : 2  
__CLASS          : __PARAMETERS  
__SUPERCLASS     :  
__DYNASTY        : __PARAMETERS  
__RELPATH        :  
__PROPERTY_COUNT : 1  
__DERIVATION     : {}  
__SERVER         :  
__NAMESPACE      :  
__PATH           :  
ReturnValue      : 0  
PSCoordinateIndex :  
  
Ln 22 Col 25 | 100%
```

Tip

You might notice that the preceding screenshot is an elevated session. If you try to start a service in a non-elevated session, you get a `ReturnValue` of 2, which means "warning".

WMI and CIM

PowerShell 3.0 introduced the **Common Information Model (CIM)** cmdlets to access WMI repositories. The WMI cmdlets are still present, but they have been superseded by the CIM cmdlets that can perform all the same functions. From a functional level, the CIM cmdlets are similar to the WMI cmdlets. The main difference is the communication that the cmdlets perform with the target computer. The WMI cmdlets use the DCOM protocol, which is a pretty standard thing to do on Windows systems, especially 20 years ago when WMI was created. Unfortunately, DCOM is kind of a mess from a firewall standpoint, as it uses a range of ports (over 16000 ports by default) that need to be open in order to function. In a data center situation, requiring this many ports to be opened in the firewalls is something that network engineers frown upon.

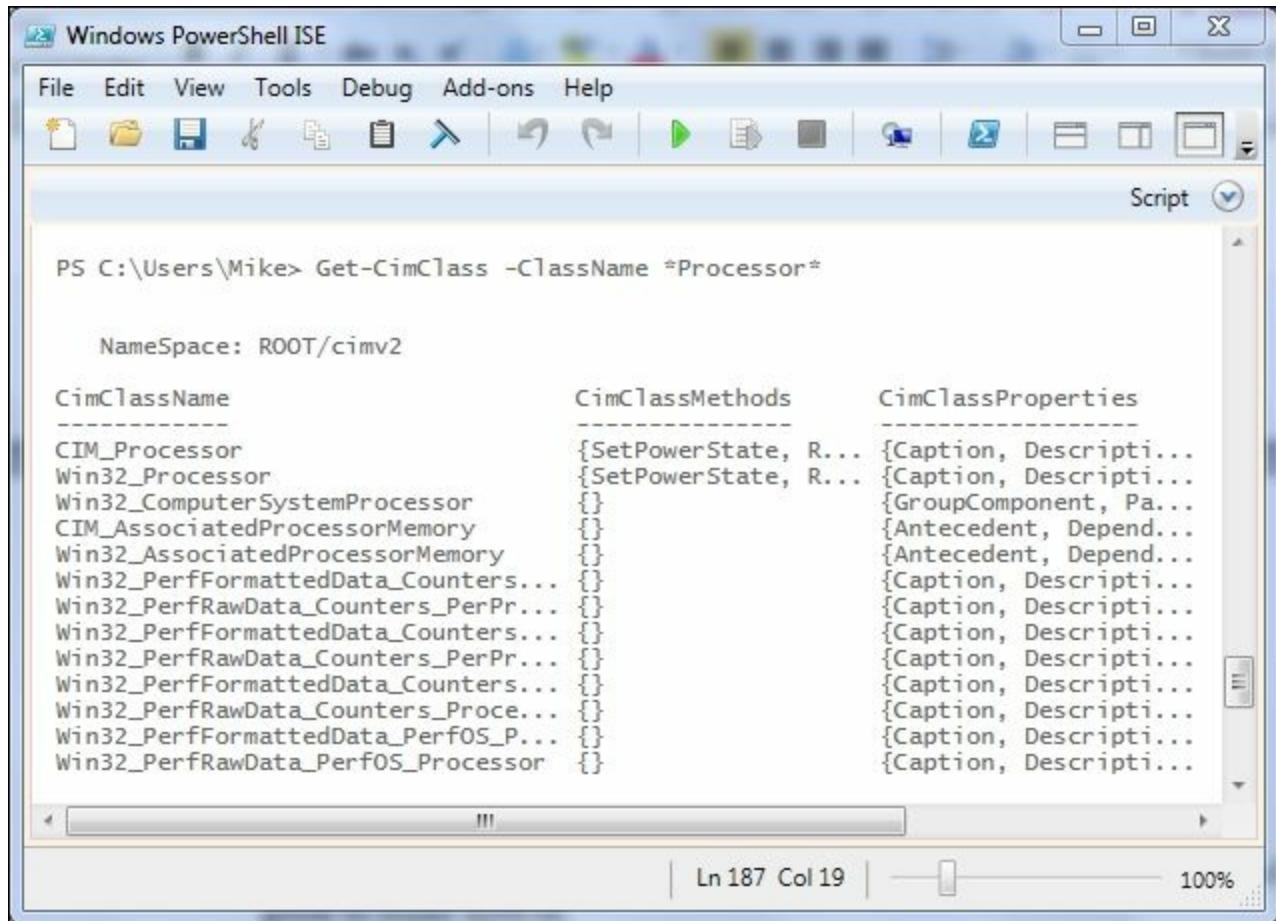
The CIM cmdlets, by default, use **Web Services for Management (WS-MAN)**, the protocol that PowerShell remoting is based on. WS-MAN uses a single port, and has a number of other properties that make it ideal from a networking standpoint. Finally, the CIM cmdlets support sessions, which allow you to reuse a connection to a computer or a set of computers for multiple requests, which can dramatically reduce the network traffic for some workloads.

If you're just looking at the WMI repositories on the local box like we've done in the examples of this chapter, both sets of cmdlets will work fine. If you're looking at the WMI classes across multiple computers in a network, you will need to consider firewall rules and whether WS-MAN is enabled. These are beyond the scope of Module 1, please see Module 2 and 3 for more, but are good to make a note of.

The CIM cmdlets

The two main CIM cmdlets that you will use are `Get-CIMClass` and `Get-CIMInstance`. Although `Get-WMIOObject` is used to retrieve both the WMI classes (with `-List`) and WMI instances (without `-List`), these roles are carried out by separate CIM cmdlets. Using `Get-CIMClass` is virtually

identical to using `Get-WMIObject` with the `-List` switch. We can recreate the processor example from earlier in the chapter with `Get-CIMClass` as follows:



The screenshot shows a Windows PowerShell ISE window. The command entered is `Get-CimClass -ClassName *Processor*`. The output displays a table with three columns: CimClassName, CimClassMethods, and CimClassProperties. The CimClassName column lists various processor-related classes. The CimClassMethods and CimClassProperties columns show empty or partially visible lists of methods and properties respectively for each class.

CimClassName	CimClassMethods	CimClassProperties
CIM_Processor	{SetPowerState, R...	{Caption, Descripti...
Win32_Processor	{SetPowerState, R...	{Caption, Descripti...
Win32_ComputerSystemProcessor	{}	{GroupComponent, Pa...
CIM_AssociatedProcessorMemory	{}	{Antecedent, Depend...
Win32_AssociatedProcessorMemory	{}	{Antecedent, Depend...
Win32_PerfFormattedData_Counters...	{}	{Caption, Descripti...
Win32_PerfRawData_Counters_PerPr...	{}	{Caption, Descripti...
Win32_PerfFormattedData_Counters...	{}	{Caption, Descripti...
Win32_PerfRawData_Counters_PerPr...	{}	{Caption, Descripti...
Win32_PerfFormattedData_Counters...	{}	{Caption, Descripti...
Win32_PerfRawData_Counters_Proce...	{}	{Caption, Descripti...
Win32_PerfFormattedData_PerfOS_P...	{}	{Caption, Descripti...
Win32_PerfRawData_PerfOS_Processor	{}	{Caption, Descripti...

Retrieving instances is done, of course, with `Get-CIMInstance`. Filters work exactly like they do with `Get-WMIObject`. Also, the objects output by these two cmdlets are the same, so working with the results will be exactly the same.

CDXML modules

One of the drawbacks of using the WMI or CIM cmdlets is that you don't get to use parameters that are specific to the properties of the WMI class you are dealing with. For instance, when we looked at the `Win32_LogicalDisk` instances, it would have been really nice to have parameters such as `DriveType` or `Name` to easily filter the output without having to use a WQL filter statement. In PowerShell 3.0, the concept of a CDXML module was introduced. CDXML modules allow you to create an XML description of the cmdlets and the associated parameters dealing with a WMI class, and the PowerShell engine creates the cmdlets for you when you import the module.

The CDXML modules have two main advantages, as follows:

- The module is XML, so no coding is necessary.
- End users get to use parameters that are specific to the type of object rather than a generic `-Filter` parameter

In PowerShell 4.0, the majority of modules on a server will be CDXML modules, because they are easy to write.

Summary

In this chapter, we have taken a quick look at the basics of using the WMI and CIM cmdlets. It should be clear that there is a lot of information in the WMI repository, and these cmdlets are essential tools for PowerShell scripters. With the introduction of CDXML modules, we're also seeing more cmdlets that are customized to specific WMI classes, making their usage even easier.

In the final chapter, we will look at administering Internet Information Services (IIS).

For further reading

- Get-Help about_WMI
- Get-Help about_WMI_Cmdlets
- Get-Help about_WQL
- WMI Result codes: [https://msdn.microsoft.com/en-us/library/aa394574\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa394574(v=vs.85).aspx)
- Get-Help about_CIMSession
- Article summarizing CIM/WMI/OMI/DMTF terminology:
<http://powershell.org/wp/2015/04/24/management-information-the-omicimwmimidmtf-dictionary/>

Chapter 11. Web Server Administration

In this chapter, we will learn how to deal with **Internet Information Services (IIS)**, the web server that ships with Windows servers. We will specifically look at the following topics:

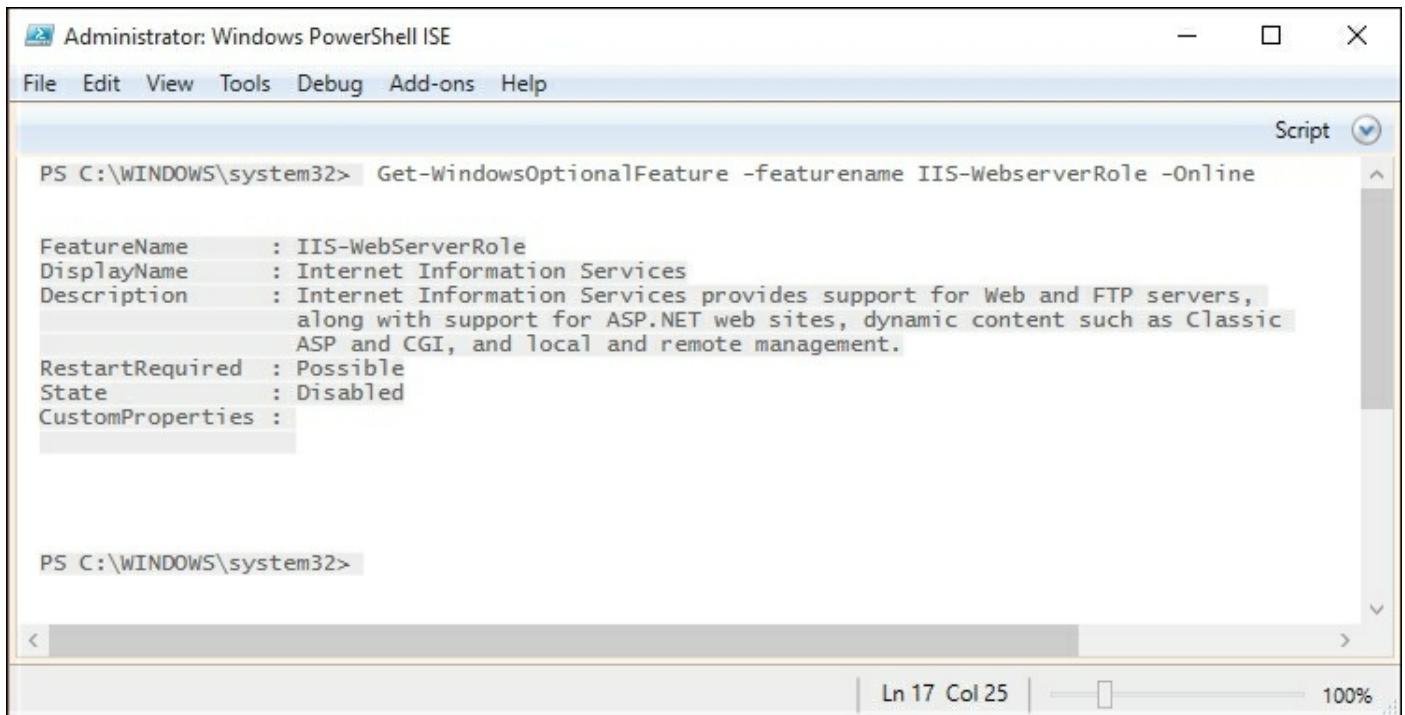
- Installing IIS
- The WebAdministration module
- Starting, stopping, and restarting IIS
- Creating virtual directories and web applications
- Working with app pools

Installing IIS

Before we install IIS, we need to determine whether it is already installed. This is done differently in a client OS, such as Windows 8.1, rather than in a server OS such as Server 2012R2.

Detecting and installing IIS in Windows 8.1

In a client OS, the cmdlet to use is `Get-WindowsOptionalFeature`, and the name of the feature to look for is `IIS-WebServerRole`. The `-Online` switch tells the cmdlet to examine the OS running on the computer rather than looking in a Windows image. In the following screenshot, you can see that IIS is not enabled on my Windows 8.1 computer:



Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

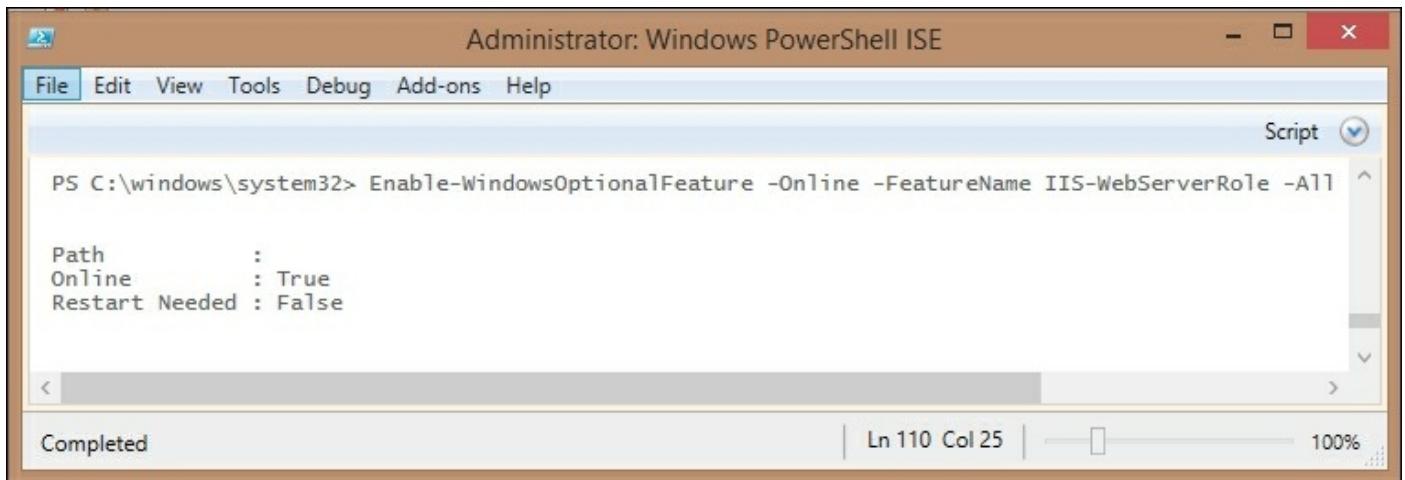
PS C:\WINDOWS\system32> Get-WindowsOptionalFeature -FeatureName IIS-WebserverRole -Online

```
FeatureName      : IIS-WebServerRole
 DisplayName     : Internet Information Services
 Description     : Internet Information Services provides support for Web and FTP servers,
                   along with support for ASP.NET web sites, dynamic content such as Classic
                   ASP and CGI, and local and remote management.
 RestartRequired : Possible
 State          : Disabled
 CustomProperties :
```

PS C:\WINDOWS\system32>

Ln 17 Col 25 | 100%

To install IIS, you can use the `Enable-WindowsOptionalFeature` cmdlet using the `IIS-WebServerRole` feature name again and the `-Online` switch. I also added the `-All` switch to tell `Enable-WindowsOptionalFeature` to enable any required features as well:



Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

PS C:\windows\system32> Enable-WindowsOptionalFeature -Online -FeatureName IIS-WebServerRole -All

```
Path          :
 Online        : True
 Restart Needed : False
```

Completed | Ln 110 Col 25 | 100%

Detecting and installing IIS in Server 2012R2

The process to detect and install IIS in Server 2012R2 is similar to what we did in Windows 8.1, but it uses slightly different cmdlets. To find out

whether IIS is enabled, we will use the `Get-WindowsFeature` cmdlet and we will specify `Web-*` as the name of the feature we're looking for:

The screenshot shows the Windows PowerShell ISE window with the title bar "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with a "Script" button is visible. The main pane displays the command "PS C:\Users\Administrator> Get-WindowsFeature -Name Web-*". The output is a table with three columns: Display Name, Name, and Install State. The table lists various IIS features with their corresponding names and install states.

Display Name	Name	Install State
[] Web Application Proxy	Web-Application-Proxy	Available
[] Web Server (IIS)	Web-Server	Available
[] Web Server	Web-WebServer	Available
[] Common HTTP Features	Web-Common-Http	Available
[] Default Document	Web-Default-Doc	Available
[] Directory Browsing	Web-Dir-Browsing	Available
[] HTTP Errors	Web-Http-Errors	Available
[] Static Content	Web-Static-Content	Available
[] HTTP Redirection	Web-Http-Redirect	Available
[] WebDAV Publishing	Web-DAV-Publishing	Available
[] Health and Diagnostics	Web-Health	Available
[] HTTP Logging	Web-Http-Logging	Available
[] Custom Logging	Web-Custom-Logging	Available
[] Logging Tools	Web-Log-Libraries	Available
[] ODBC Logging	Web-ODBC-Logging	Available
[] Request Monitor	Web-Request-Monitor	Available
[] Tracing	Web-Http-Tracing	Available
[] Performance	Web-Performance	Available

This output is organized very nicely, mimicking the Windows Feature control panel applet. You can easily see the hierarchy of features, and the state of the feature is shown in the box to the left of the display name. In our case, we want to install the `Web-Server` feature. We can do this using the `Add-WindowsFeature` cmdlet:

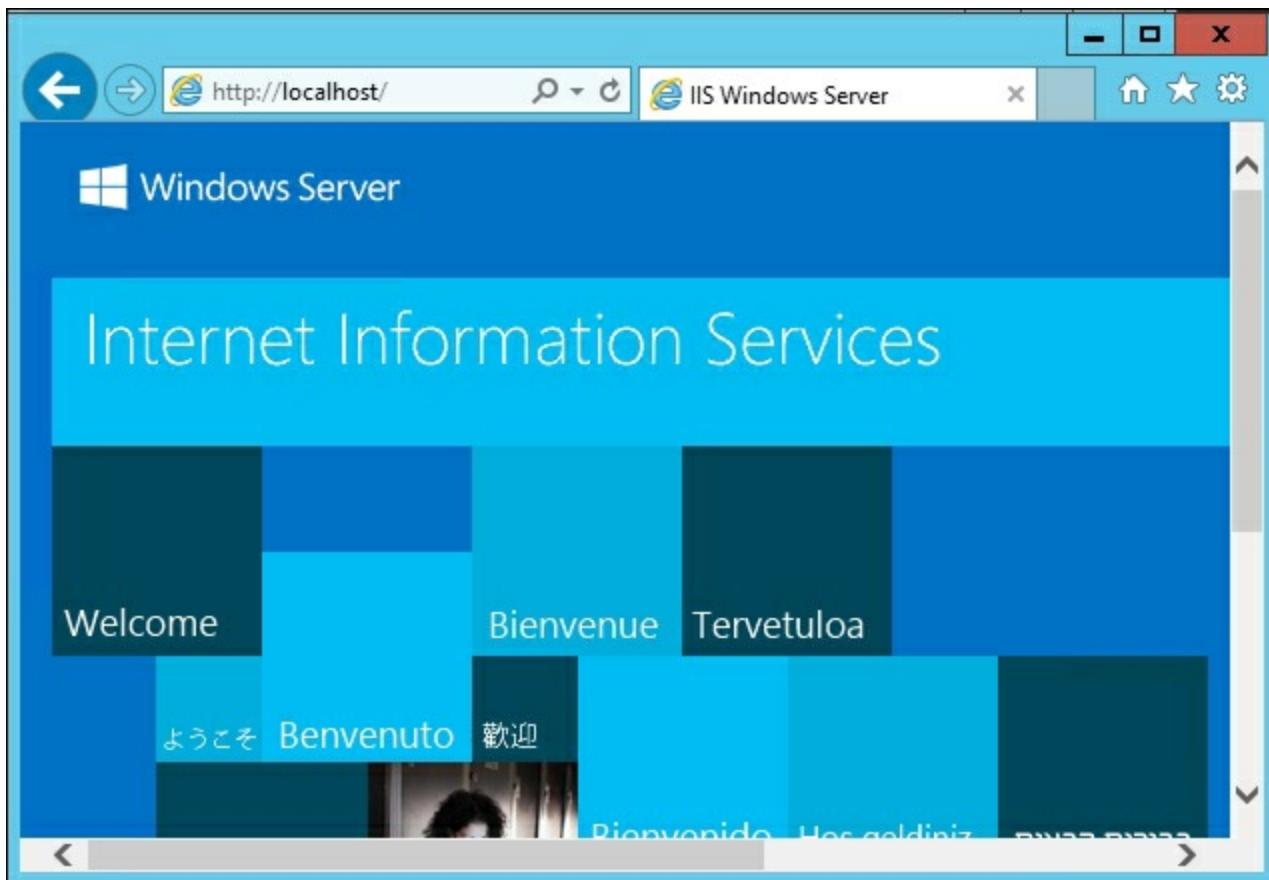
The screenshot shows the Windows PowerShell ISE window with the title bar "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with a "Script" button is visible. The main pane displays the command "PS C:\Users\Administrator> Add-WindowsFeature -Name Web-Server". The output is a table with four columns: Success, Restart Needed, Exit Code, and Feature Result. It shows the feature was added successfully with a warning about automatic updating.

Success	Restart Needed	Exit Code	Feature Result
True	No	Success	{Common HTTP Features, Default Document, D...

WARNING: Windows automatic updating is not enabled. To ensure that your newly-installed role or feature is automatically updated, turn on Windows Update.

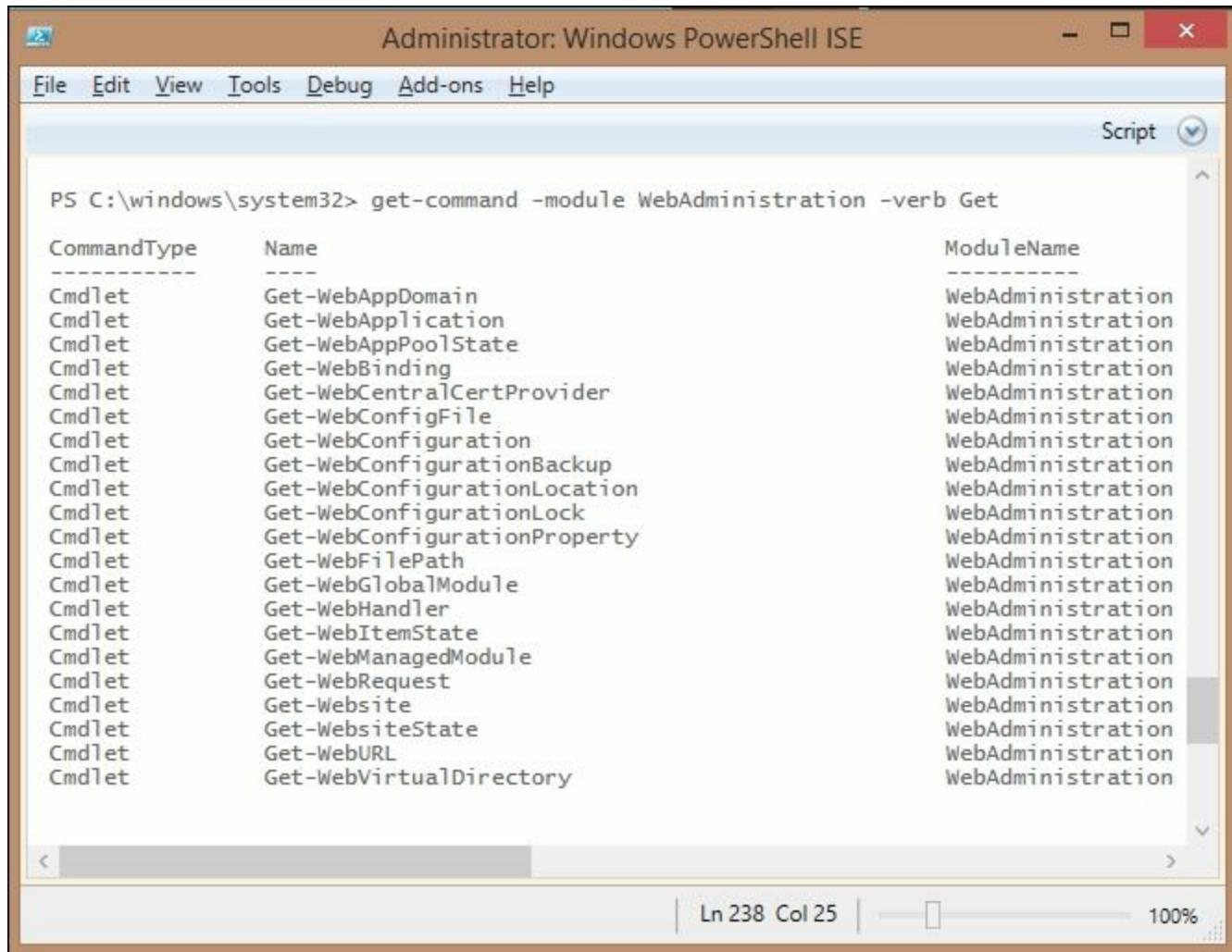
Verifying IIS

For either a client or server OS, we would verify that IIS is enabled using the associated `Get- cmdlet`, but it makes more sense to me to try to load a web page. IIS configures a default web site with a standard web page, so we should be able to navigate to `http://localhost` and know immediately whether everything is working well:



The WebAdministration module

Once IIS is installed, we need to import the `WebAdministration` module to interact with IIS (using `Import-Module WebAdministration`). Using the `Get-Command` cmdlet to find the `Get-` cmdlets in the module is a good way to get an idea about what the module allows us to work with:



A screenshot of the Windows PowerShell ISE window. The title bar says "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu shows "Script" selected. The command entered in the console is "PS C:\windows\system32> get-command -module WebAdministration -verb Get". The output is a table with three columns: CommandType, Name, and ModuleName. The CommandType column shows all entries as Cmdlets. The Name column lists various cmdlets such as Get-WebAppDomain, Get-WebApplication, Get-WebAppPoolState, etc. The ModuleName column shows all entries as "WebAdministration".

CommandType	Name	ModuleName
Cmdlet	Get-WebAppDomain	WebAdministration
Cmdlet	Get-WebApplication	WebAdministration
Cmdlet	Get-WebAppPoolState	WebAdministration
Cmdlet	Get-WebBinding	WebAdministration
Cmdlet	Get-WebCentralCertProvider	WebAdministration
Cmdlet	Get-WebConfigFile	WebAdministration
Cmdlet	Get-WebConfiguration	WebAdministration
Cmdlet	Get-WebConfigurationBackup	WebAdministration
Cmdlet	Get-WebConfigurationLocation	WebAdministration
Cmdlet	Get-WebConfigurationLock	WebAdministration
Cmdlet	Get-WebConfigurationProperty	WebAdministration
Cmdlet	Get-WebFilePath	WebAdministration
Cmdlet	Get-WebGlobalModule	WebAdministration
Cmdlet	Get-WebHandler	WebAdministration
Cmdlet	Get-WebItemState	WebAdministration
Cmdlet	Get-WebManagedModule	WebAdministration
Cmdlet	Get-WebRequest	WebAdministration
Cmdlet	Get-Website	WebAdministration
Cmdlet	Get-WebsiteState	WebAdministration
Cmdlet	Get-WebURL	WebAdministration
Cmdlet	Get-WebVirtualDirectory	WebAdministration

Here, we can see that we have cmdlets to deal with sites, virtual directories, applications, bindings, configuration, and much more. Another thing that is worth mentioning at this point is that the `WebAdministration` module also adds a new PSProvider called `WebAdministration` and a PSDrive called `IIS`, which exposes a hierarchical view of the IIS installation. Remember that PSDrives are how PowerShell exposes hierarchical data. In this case, the `IIS`

configuration is treated in a similar way to a drive.

Tip

You try it!

Starting with `CD IIS:`, explore the IIS installation using the `DIR (Get-ChildItem)` cmdlet. Don't forget to change back to a filesystem drive when you're done.

Starting, stopping, and restarting IIS

Since you can run command-line programs in PowerShell, the `IISReset` command can be used to start, stop, and restart IIS using the `/START`, `/STOP`, and `/RESTART` switches:



The screenshot shows an "Administrator: Windows PowerShell" window. The title bar says "Administrator: Windows PowerShell". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar has a "Script" button. The main area shows the command PS C:\> iisreset /restart followed by its output: "Attempting stop...", "Internet services successfully stopped", "Attempting start...", and "Internet services successfully restarted". The status bar at the bottom shows "Completed | Ln 367 Col 9 | 100%".

If you want to start or stop a particular website rather than the entire IIS installation, you need to use the `Start-WebSite` and `Stop-WebSite` cmdlets. They both have a `-Name` parameter that allows you to specify which site you want to work with. In the following screenshot, I am stopping and starting a website called `Test`. Also, I have used the `Get-WebSite` cmdlet after each step to show that the `Test` site stopped and started correctly:

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script

```
PS C:\WINDOWS\system32> stop-website -Name Test
PS C:\WINDOWS\system32> Get-Website
Name          ID  State       Physical Path           Bindings
----          --  -----      -----          -----
Default Web Site 1  1    Started     %SystemDrive%\inetpub\wwwroot  http *:80:
Test          2    Stopped     c:\temp                  http *:8080:

PS C:\WINDOWS\system32> Start-WebSite -name Test
PS C:\WINDOWS\system32> Get-Website
Name          ID  State       Physical Path           Bindings
----          --  -----      -----          -----
Default Web Site 1  1    Started     %SystemDrive%\inetpub\wwwroot  http *:80:
Test          2    Started     c:\temp                  http *:8080:
```

< >

Ln 23 Col 25 | 100%

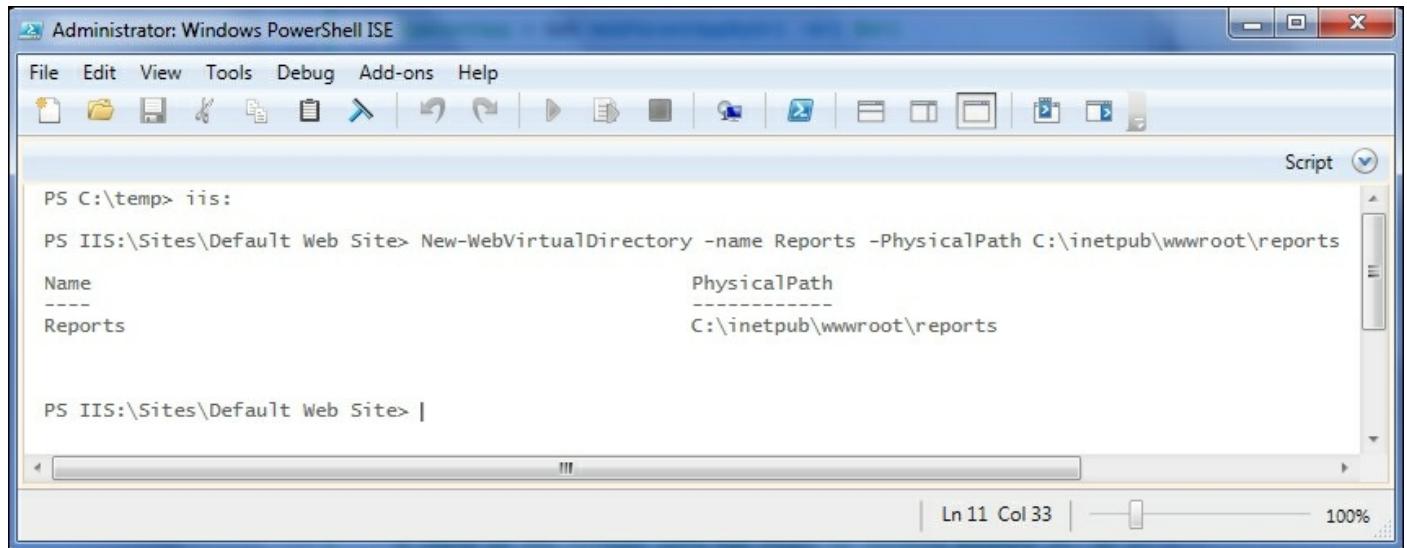
Creating virtual directories and web applications

Virtual directories and web applications are the different options to contain content in IIS. A virtual directory is a pointer to a location on the disk where the content actually resides. A web application, in the IIS terminology, is a virtual directory that also has the ability to run in a different worker process than its parent.

To create a virtual directory, we will use the `New-WebVirtualDirectory` cmdlet and supply the `-Name` and `-PhysicalPath` parameters. Also, we will need to specify the site and we can do this in one of the following two ways:

1. Use `Set-Location (CD)` in the IIS drive and navigate to the desired site.
2. Specify the site on the command line.

In the following screenshot, we will illustrate the first method:



The screenshot shows a Windows PowerShell ISE window titled "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations. The main pane displays PowerShell commands and their output:

```
PS C:\temp> iis:
PS IIS:\Sites\Default Web Site> New-WebVirtualDirectory -name Reports -PhysicalPath C:\inetpub\wwwroot\reports
Name          PhysicalPath
----          -----
Reports      C:\inetpub\wwwroot\reports

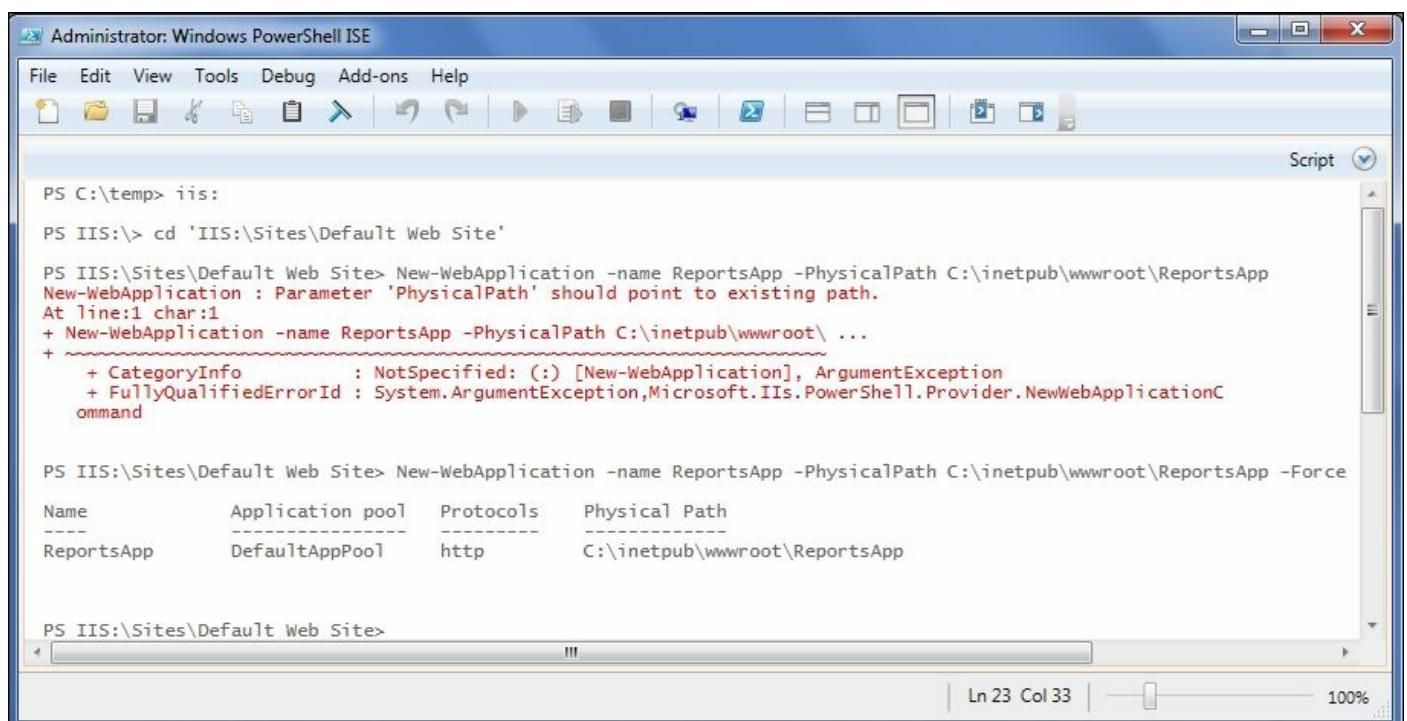
PS IIS:\Sites\Default Web Site> |
```

The status bar at the bottom indicates "Ln 11 Col 33" and "100%".

If you are working with a website besides the default site (`Default Web Site`), you will need to set the location to this site. For instance, to

create a virtual directory in the Test site, you would use Set-Location IIS:\Sites\Test\.

The process to create a web application is similar. The New-WebApplication cmdlet takes -Name and -PhysicalPath parameters as well as a -Site parameter that can be supplied in the same fashion as with New-WebVirtualDirectory. Here, we create a new web application called ReportsApp in the root of the default web site. Note that the cmdlet expects the physical path to be an existing directory, or you can use the -Force switch to make the cmdlet create the folder for you. I've tried to create a web application with a path that doesn't exist, as shown in the following screenshot. After receiving an error, I retried with -Force and was successful.



The screenshot shows a Windows PowerShell ISE window with the title "Administrator: Windows PowerShell ISE". The command PS C:\temp> iis: is run, followed by PS IIS:> cd 'IIS:\Sites\Default Web Site'. Then, PS IIS:\Sites\Default Web Site> New-WebApplication -name ReportsApp -PhysicalPath C:\inetpub\wwwroot\ReportsApp is run, which fails with the error "New-WebApplication : Parameter 'PhysicalPath' should point to existing path." At line:1 char:1 + New-WebApplication -name ReportsApp -PhysicalPath C:\inetpub\wwwroot\ ... + CategoryInfo : NotSpecified: (:) [New-WebApplication], ArgumentException + FullyQualifiedErrorId : System.ArgumentException,Microsoft.IIs.PowerShell.Provider.NewWebApplicationCommand. The command is then retried with the -Force switch: PS IIS:\Sites\Default Web Site> New-WebApplication -name ReportsApp -PhysicalPath C:\inetpub\wwwroot\ReportsApp -Force. The output shows the new application has been created with the following properties:

Name	Application pool	Protocols	Physical Path
ReportsApp	DefaultAppPool	http	C:\inetpub\wwwroot\ReportsApp

If we wanted the new application to run in a specific application pool, we would have used the -ApplicationPool parameter to specify its name.

You can easily see the virtual directories and web applications with the IIS drive. After using CD (Set-Location) in the \Sites\Default Web Site folder, DIR (Get-ChildItem) shows all of the folders, files, virtual

directories, and web applications:

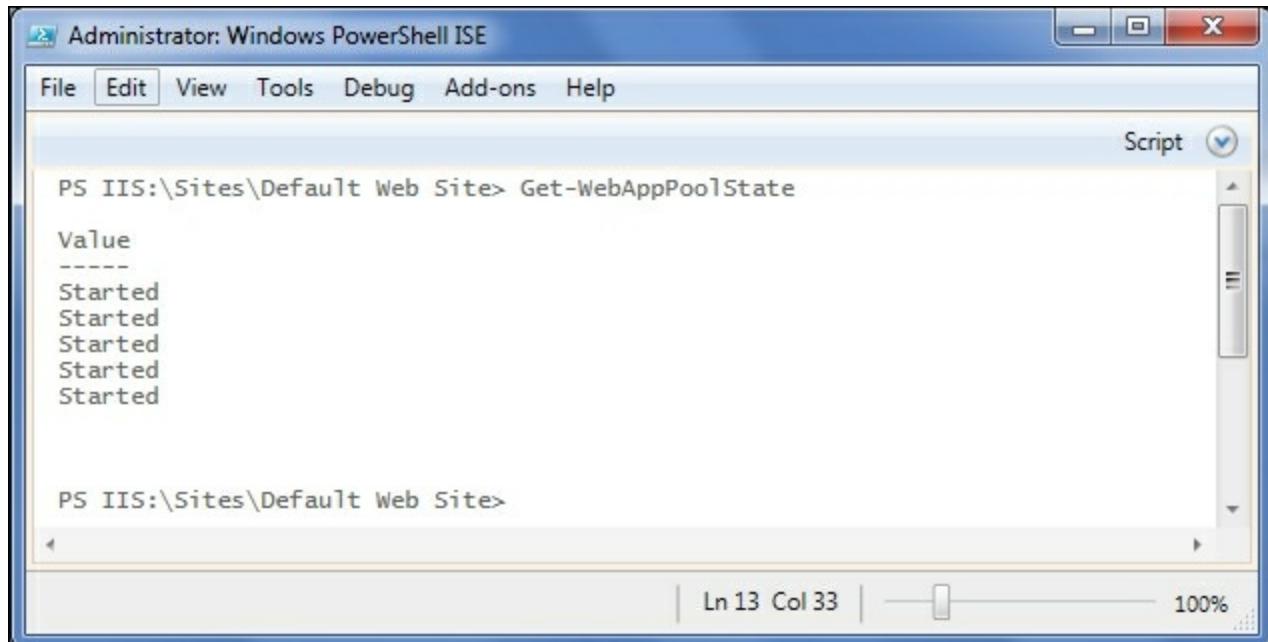
The screenshot shows a Windows PowerShell ISE window titled "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu on the right says "Script". The main pane displays the output of the command "PS IIS:\Sites\Default Web Site> dir". The output is a table with columns: Type, Name, and Physical Path. The table lists the following items:

Type	Name	Physical Path
directory	aspnet_client	C:\inetpub\wwwroot\aspnet_client
file	iisstart.htm	C:\inetpub\wwwroot\iisstart.htm
file	Process.html	C:\inetpub\wwwroot\Process.html
virtualDirectory	Reports	C:\inetpub\wwwroot\reports
application	ReportsApp	C:\inetpub\wwwroot\ReportsApp
file	welcome.png	C:\inetpub\wwwroot\welcome.png

Below the table, the command "PS IIS:\Sites\Default Web Site> |" is visible, followed by a scroll bar, and the status bar at the bottom shows "Ln 36 Col 33" and "100%".

Working with application pools

Given the fantastic cmdlet support for virtual directories and web applications, I was surprised to find that there isn't a `Get-WebAppPool` cmdlet. There is a `Get-WebAppPoolState` cmdlet, but the formatted output isn't particularly useful.



The screenshot shows a Windows PowerShell ISE window titled "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar has a "Script" dropdown. The main pane displays the command "PS IIS:\Sites\Default Web Site> Get-WebAppPoolState" followed by the output:

```
Value
-----
Started
Started
Started
Started
Started
```

The bottom status bar shows "Ln 13 Col 33" and "100%".

From the previous screenshot, you can see that there are five application pools and they have all been started, but you don't know what they are called. If one showed **Stopped**, for instance, you wouldn't know which one you needed to start. Adding `Select-Object -Property *` helps sometimes, but the values aren't easy to use.

The screenshot shows a Windows PowerShell Integrated Scripting Environment (ISE) window titled "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A dropdown menu labeled "Script" is open. The main pane displays the following PowerShell command and its output:

```
PS IIS:\Sites\Default Web Site> Get-WebAppPoolState | select-object -property *
```

```
ItemXPath      : /system.applicationHost/applicationPools/add[@name='ASP.NET  
1.1']  
MemberType     : CodeProperty  
IsSettable     : False  
IsGettable     : True  
Value          : Started  
TypeNameOfValue: System.UInt32  
Name           : state  
IsInstance     : True  
  
ItemXPath      : /system.applicationHost/applicationPools/add[@name='ASP.NET  
v4.0']  
MemberType     : CodeProperty  
IsSettable     : False  
IsGettable     : True  
Value          : Started  
TypeNameOfValue: System.UInt32  
Name           : state  
IsInstance     : True
```

The status bar at the bottom indicates "Ln 63 Col 24" and "100%".

Since the name of the application pool is embedded in an XPath expression, it is not very easy to work with. Fortunately for us, the application pools are easy to find in the IIS drive, so we can craft our own function to return the app pools.

The screenshot shows a Windows PowerShell ISE window. At the top, the title bar reads "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A tab labeled "GetWebAppPool.ps1" is open. The script content is:

```
1 function Get-WebAppPool{
2     param([string]$name)
3     if($name){
4         dir IIS:\AppPools | where Name -like $name
5     } else {
6         dir IIS:\AppPools
7     }
8 }
```

Below the script, the command "PS IIS:\Sites\Default Web Site> Get-WebAppPool" is run, followed by its output:

Name	State	Applications
ASP.NET 1.1	Started	
ASP.NET v4.0	Started	
ASP.NET v4.0 Classic	Started	
Classic .NET AppPool	Started	
DefaultAppPool	Started	Default Web Site /ReportsApp

Then, the command "PS IIS:\Sites\Default Web Site> Get-WebAppPool Default*" is run, followed by its output:

Name	State	Applications
DefaultAppPool	Started	Default Web Site /ReportsApp

The status bar at the bottom left says "Completed", the cursor position is "Ln 27 Col 47", and the zoom level is "100%".

Creating application pools

We can create an app pool using the `New-WebAppPool` cmdlet, which only has one interesting parameter called `-Name`. We're going to create an app pool called `ReportPool` and later configure the `ReportApp` web application to run in this app pool:

```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script

PS IIS:\Sites\Default Web Site> New-WebAppPool -Name ReportPool
Name          State      Applications
----          -----      -----
ReportPool    Started
```

The screenshot shows a Windows PowerShell ISE window with the title bar "Administrator: Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A "Script" tab is selected. The command "New-WebAppPool -Name ReportPool" is run, and the output shows a table with one row: "ReportPool" in the Name column, "Started" in the State column, and nothing in the Applications column.

Switching an application to run in this pool involves a PSProvider-related cmdlet called `Set-ItemProperty`:

```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script

PS IIS:\Sites\Default Web Site> Set-ItemProperty '.\ReportsApp' ApplicationPool ReportPool
PS IIS:\Sites\Default Web Site>
```

The screenshot shows a Windows PowerShell ISE window with the title bar "Administrator: Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A "Script" tab is selected. The commands "Set-ItemProperty '.\ReportsApp' ApplicationPool ReportPool" and "Set-ItemProperty '.\ReportsApp' ApplicationPool ReportPool" are run, and the output shows the prompt "PS IIS:\Sites\Default Web Site>" again.

We can easily verify that the changes worked using our `Get-WebAppPool` function from earlier in the chapter:

```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script

PS IIS:\Sites\Default Web Site> Get-WebAppPool -name ReportPool
Name          State      Applications
----          -----      -----
ReportPool    Started    /ReportsApp
```

The screenshot shows a Windows PowerShell ISE window with the title bar "Administrator: Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". A "Script" tab is selected. The command "Get-WebAppPool -name ReportPool" is run, and the output shows a table with one row: "ReportPool" in the Name column, "Started" in the State column, and "/ReportsApp" in the Applications column.

Summary

In this chapter, we saw how to work with IIS, including the installation and validation. We created virtual directories, web applications, and application pools, and learned how to use the IIS drive.

For further reading

- Get-Help Get-WindowsOptionalFeature
- Get-Help Enable-WindowsOptionalFeature
- Get-Help Get-WindowsFeature
- Get-Help Add-WindowsFeature
- The WebAdministration module: <https://technet.microsoft.com/en-us/library/ee790599.aspx>
- Get-help Start-Website
- Get-help Stop-Website
- Get-help Get-Website
- Get-help New-WebVirtualDirectory
- Get-help New-WebApplication
- Get-help Get-WebAppPoolState
- Get-help New-WebAppPool
- Get-help Set-ItemProperty

Appendix A. Next Steps

The goal of Module 1 has been to get you started with PowerShell. To this end, we have looked at several broad topics, such as:

- Navigating PowerShell with `Get-Command`, `Get-Help`, and `Get-Member`
- Using the Pipeline to combine the commands
- Packaging code in scripts, functions, and modules
- Interacting with files and WMI
- Administering IIS

These skills will take you far in PowerShell, but this is just the beginning of what you can do. Here, I have collected some suggested "next steps" that you will find useful as skills to add to your PowerShell repertoire:

- Explore advanced functions, such as:
 - Using common parameters
 - Pipeline the input
 - Parameter validation
- PowerShell Remoting
- PowerShell Workflows
- Desired State Configuration
- Explore miscellaneous PowerShell topics, including the following:
 - Working with .NET and COM objects
 - Working with SQL Server data
 - Building a GUI
- Administering Microsoft systems such as:
 - Exchange
 - SQL Server
 - System Center

You should have no trouble finding resources to guide you through any of these topics. If you are stuck, the PowerShell community on StackOverflow, PowerShell.org (<http://powershell.org/wp/>), and reddit are very strong and encouraging to scripters at all levels of expertise.

Part 2. Module 2

Windows PowerShell for .NET Developers - Second Edition

Efficiently administer and maintain your development environment with Windows PowerShell

Chapter 1. Getting Started with Windows PowerShell

In this chapter, we will cover the following topics:

- What is **Windows PowerShell**?
- Installing **Windows Management Framework 5.0** on **Windows Server 2012**
- The **Windows PowerShell console**
- Setting up the console using GUI and PowerShell
- Benefits of the **Windows PowerShell ISE**
- Creating snippets in the PowerShell ISE

Scripting the cmdlet style

This is a self-paced guide for you to learn Windows PowerShell in order to perform IT automation. The management goals of IT are to simplify the creation and operation of computing environments. IT professionals and developers need to complete their daily tasks swiftly in order to avoid chaos.

Windows PowerShell is an object-based and distributed automation platform that enables IT professionals and developers to speed up their development tasks and deployments. The Windows PowerShell scripting language makes daily tasks more productive.

The goal of Module 2 is to share the features of Windows PowerShell 5.0 and how developers can utilize its benefits. Let's dive into Windows PowerShell 5.0 so that we can design our infrastructure, automate our daily tasks, manage drifts in our environment, quicken the deployment tasks, and so on.

The IT industry is growing rapidly; to adapt and minimize the impact, we need to plan and design our infrastructure so as to meet the growing needs with minimum chaos. It's challenging and really difficult to

automate the tasks without PowerShell. Windows PowerShell plays a major role by not only enabling task automation, but also allowing us to design our infrastructure with the help of IT professionals and developers.

Developers from the .NET background can quickly understand PowerShell. Developers can easily create their own **cmdlets** (cmdlets, pronounced as command lets, are nothing but the lightweight commands used in Windows PowerShell) and leverage it in the infrastructure as needed. Thus, the deployment tasks become easier and the productivity increases by automation.

Come, let's dive into Windows PowerShell 5.0 and its features.

Note

Before we begin, let's note that Windows PowerShell 5.0 is not a stable release. So, if you identify any bugs in the command, you can file a case at <http://connect.microsoft.com>.

This self-paced guide will discuss the following:

- Working with PowerShell cmdlets
- PowerShell scripting
- Exploring PowerShell modules
- Exploring the XML, COM, and .NET objects
- Exploring JSON, REST API, and Web Services
- Features of Windows PowerShell 5.0
- Exploring DSC to configure servers
- Exploring web technologies using PowerShell
- Consuming API in C# and PowerShell
- Using PowerShell codes in C#

Note

If you are interested in knowing the origin of Windows PowerShell 5.0, you should read Monad Manifesto at:

http://blogs.msdn.com/cfs-file.ashx/_key/communityserver-

[components-postattachments/00-01-91-05-67/Monad-Manifesto-
2D00-Public.doc](#)

Introducing Windows PowerShell

Windows PowerShell is an object-based command-line interface with a very powerful scripting language built on .NET. Windows PowerShell is designed with access to several cmdlets, functions, filters, scripts, aliases, and executables.

From Windows Management Framework 4.0 onward, a new configuration management platform has been introduced in PowerShell 4.0, which is **Windows PowerShell Desired State Configuration (DSC)**.

Windows PowerShell is very enhanced in the version 5.0 April 2015 Preview release. This is not a stable release, but we will explore the new features introduced in Windows PowerShell 5.0. The current stable version is Windows PowerShell 4.0.

IT professionals and developers use their own statements or definitions based on the way they use PowerShell. Mostly, they define PowerShell as a scripting language. Yeah! This is true but not completely because Windows PowerShell does much more. However, the official page of Windows PowerShell defines PowerShell as an automation platform and scripting language for Windows and Windows Server that allows you to simplify the management of your systems. Unlike the other text-based shells, PowerShell harnesses the power of the .NET Framework, providing us with rich objects and a massive set of built-in functionality to help you take control of your Windows environments. Refer to the following URL:

<https://msdn.microsoft.com/en-us/powershell>

You may wonder, why Windows PowerShell? Why not VBScript? This is a very common question that most administrators think. It's good to know both, but VBScript is fading, and compared to PowerShell, it is very limited. Windows PowerShell can access all .NET libraries, which helps us explore DLLs easily; on the other hand, VBScript has

limitations. If you are a beginner Windows PowerShell will be a better choice because it is easier to understand and very powerful. The Windows PowerShell cmdlets are rich and very easy to understand. Using Windows PowerShell, we can avoid repeated steps, automate administration tasks, avoid GUI clicks, and do much more. This is not limited to IT Professionals; developers can also do much more by exploring Windows PowerShell, such as performing development tasks faster than before.

From Windows PowerShell 5.0 onward, we can develop classes, parse structured objects using the new cmdlets, and do the package management easily using the `PackageManagement` module. This module is introduced in version 5.0 but was formerly known as the `OneGet` module; many such modules are enhanced in the new version.

Note

But wait, this is not all; you can refer to the following link to get a list of the new features in Windows PowerShell 5.0:

https://technet.microsoft.com/en-us/library/hh857339.aspx#BKMK_new50

Installing Windows Management Framework 5.0

Before we explore the PowerShell console host and the PowerShell ISE, let's install Windows Management Framework 5.0 April 2015 Preview (Latest Build). Windows Management Framework 5.0 is now supported on Windows 7 and Windows Server 2008 R2 SP1—this will be covered in [Chapter 3, Exploring Desired State Configuration](#) in detail. Windows Management Framework is shipped out of the box in the Windows 10 operating system and Windows Server Technical Preview. So, there is no need for manual installation.

To start the demo, we need to perform the following functions in our machine environment:

- Set up Windows 2012 R2 Data Center
- Download .NET Framework 4.5 (Included in Windows Server 2012)
- Download Windows Management Framework 4.0
- Download Windows Management Framework 5.0

Note

Windows 2008 R2 has PowerShell 2.0 by default. To install Windows Management Framework 5.0, we need to upgrade from Windows Management Framework 4.0.

To open Windows PowerShell, perform the following functions:

1. Click on the start icon.
2. Type `Windows PowerShell` in **Search programs and files**.
3. It lists **Windows PowerShell ISE** and **Windows PowerShell**.

Here, ISE is short for Integrated Scripting Environment.

Note

Ignore x86—it's a 32-bit Windows PowerShell application (we will not use it in any of our examples in Module 2).

It's best practice to know your PowerShell version before you start doing an exercise or start using scripts from the Internet.

To check the current version of PowerShell, you can simply run the following code:

```
$PSVersionTable
```

The following screenshot illustrates the output:

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Untitled1.ps1* X

1 \$PSVersionTable|

```
PS C:\Users\ChenV> $PSVersionTable
Name            Value
----            -----
PSVersion       4.0
WSManStackVersion 3.0
SerializationVersion 1.1.0.1
CLRVersion      4.0.30319.34014
BuildVersion     6.3.9600.17400
PSCompatibleVersions {1.0, 2.0, 3.0, 4.0}
PSRemotingProtocolVersion 2.2

PS C:\Users\ChenV>
```

Windows Server 2012 R2 has WMF Version 4.0 by default. Let's upgrade it to WMF 5.0. If you are performing this task on Windows 7 or Windows 2008 R2 with SP1, install WMF 4.0 prior to WMF 5.0.

Following are the links where you can download the .NET framework and WMF from:

- .NET Framework 4.5—<http://www.microsoft.com/en-us/download/details.aspx?id=30653>
- WMF 5.0—<http://www.microsoft.com/en-us/download/details.aspx?>

[id=46889](#)

Note

Do read the system requirements before proceeding into the production environment. The installation of WMF 5.0 requires a reboot.

Once the .NET framework 4.0 is in place, update it to the .NET framework 4.5.

As per the Microsoft Document, the .NET framework 4.5 is included in Windows 8 and Windows Server 2012. We can skip the .NET framework 4.5 installation.

We need to identify and install the correct package of WMF 5.0. For Windows Server 2012, we need to download the 64-bit version of it, which is WindowsBlue-KB3055381-x64.msu.

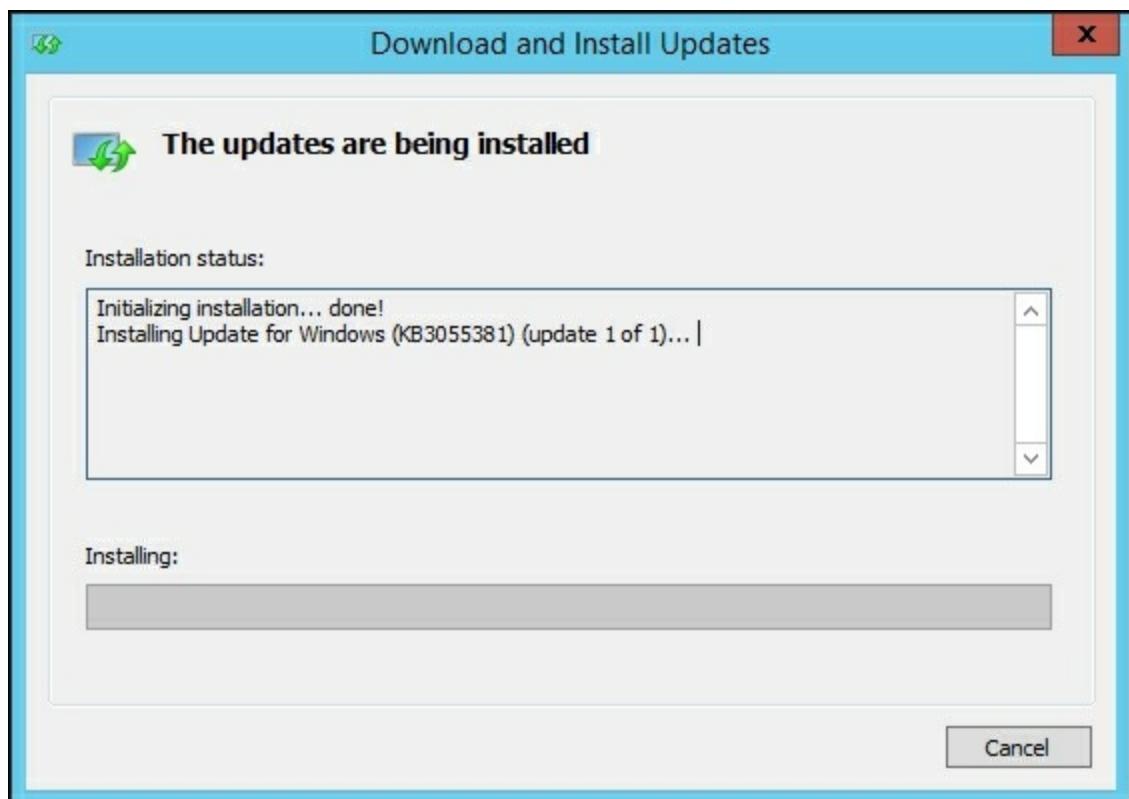
Open the MSU package and you will see the prompt, as shown in the following screenshot:



1. Click on Yes.
2. Read and accept the license agreement, as shown in the following screenshot:

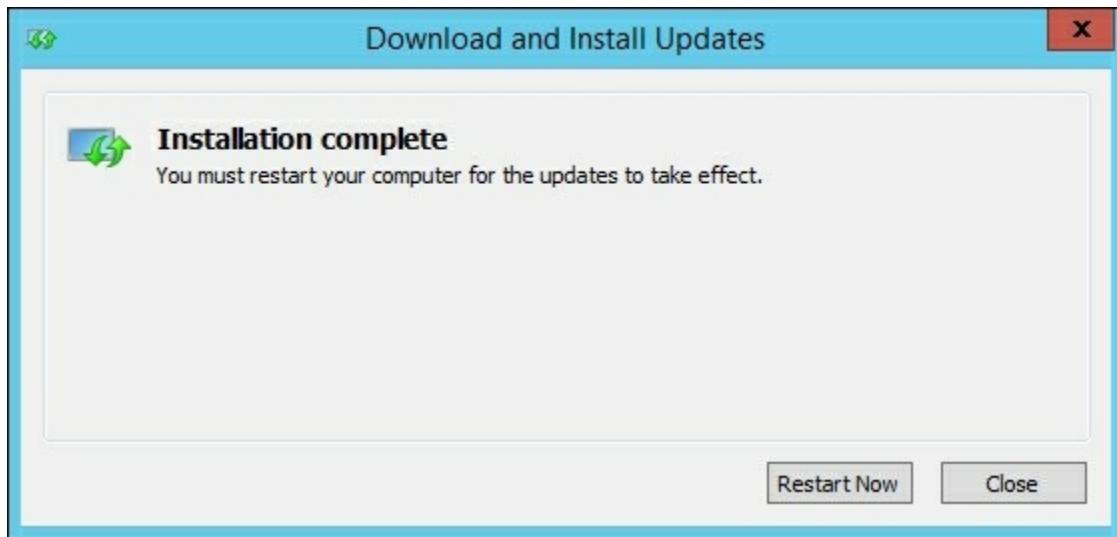


3. After this, the installation begins, as shown in the following screenshot:



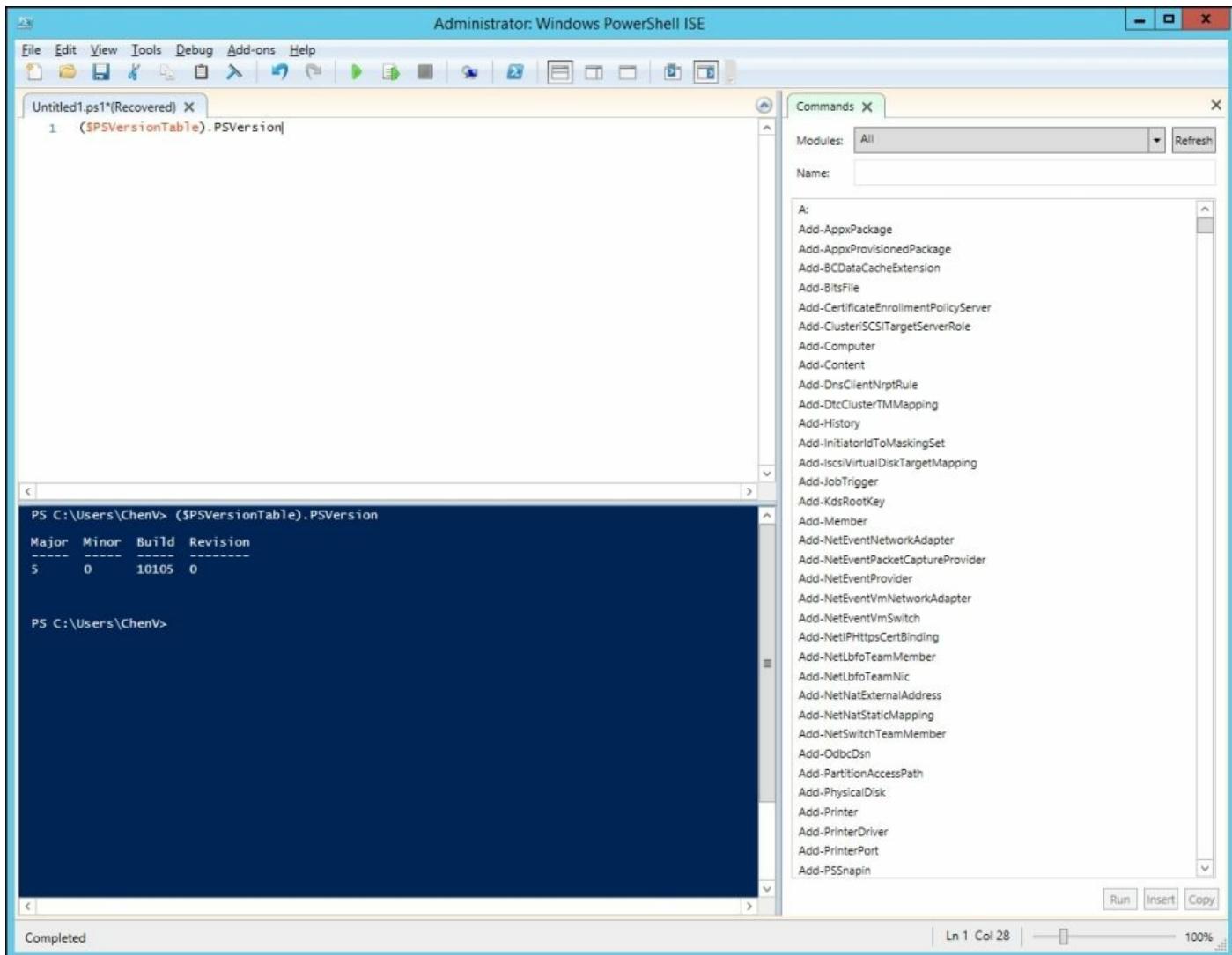
4. Click on **Restart Now** when you see the following window on your

screen:



We have successfully installed WMF 5.0. As an outcome of this upgrade, now we also have the updated versions of Windows PowerShell, Windows PowerShell **Desired State Configuration (DSC)**, and the Windows PowerShell ISE. Package manager and network switches cmdlets are included with this version.

The following image illustrates the Windows PowerShell 5.0 ISE:



The Windows PowerShell consoles

Windows PowerShell has two different consoles: the console host and the Windows PowerShell ISE host, which is GUI. To verify this, click on the start icon and search for `PowerShell` on Windows Server 2012.

The following image illustrates the results of PowerShell:



Note

Ignore x86 in the Windows PowerShell ISE (x86) for now. This is a 32-bit ISE, and throughout Module 2, we will focus only on the 64-bit ISE.

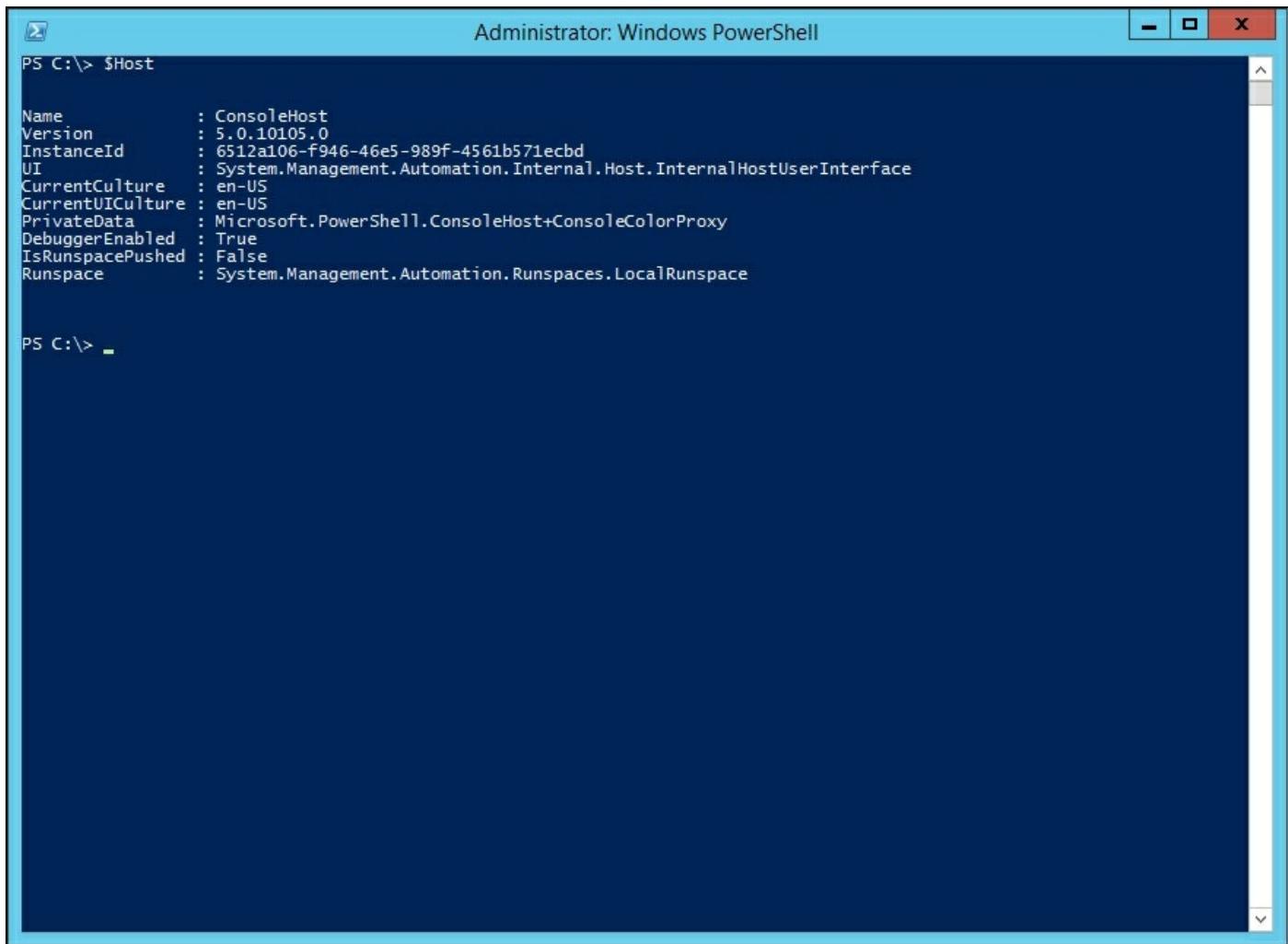
The location of the Windows PowerShell console host and ISE is:

C:\Windows\System32\WindowsPowerShell\v1.0

The file name `powershell.exe` is the console host and `powershell_ise.exe` is the Integrated Scripting Environment.

The Windows PowerShell console host

This is where we begin with the PowerShell cmdlets. It's easy for on-the-fly cmdlet executions. The following image illustrates the PowerShell 5.0 console host:



The screenshot shows an Administrator Windows PowerShell window. The title bar reads "Administrator: Windows PowerShell". The command "PS C:\> \$Host" is entered at the prompt, followed by its output:

```
Name      : ConsoleHost
Version   : 5.0.10105.0
InstanceId : 6512a106-f946-46e5-989f-4561b571ecbd
UI        : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture : en-US
CurrentUICulture : en-US
PrivateData  : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
DebuggerEnabled : True
IsRunspacePushed : False
Runspace    : System.Management.Automation.Runspaces.LocalRunspace
```

PS C:\> _

By default, the background color is blue, and in the foreground, the color of the text is white.

There is no need to type out the complete command; the PowerShell console allows Tab Completion. For example, `Get-Se` + *Tab* completes the `Get-Service` command. Tab completion is not only for the commands and parameters; we can also select the properties using tab key.

To execute the following PowerShell code:

```
Get-Service -Name BITS | Select -Property Name , Status
```

Windows PowerShell allows tab completion like this:

```
Get-Ser + Tab -N + Tab BITS | sel + Tab -Pro + Tab N + Tab , s + Tab
```

There are multiple ways to start a PowerShell console: we could either use the **Run** dialog box, or type `PowerShell` in an open command-line window. These techniques allow you to pass arguments to Windows PowerShell, including the switches that control how Windows PowerShell works and the parameters that execute additional commands. For example, you can start Windows PowerShell in the no-logo mode (which means that the logo banner is turned off) using the startup command, `PowerShell -nologo`. By default, when you start Windows PowerShell via the command shell, Windows PowerShell runs and then exits. If you want Windows PowerShell to execute a command and not terminate, type `PowerShell /noexit`, followed by the command text.

We can use this to schedule our script in task schedulers. To see all the switches, run the following command:

```
PowerShell.exe /?
```

Setting up the console host using GUI

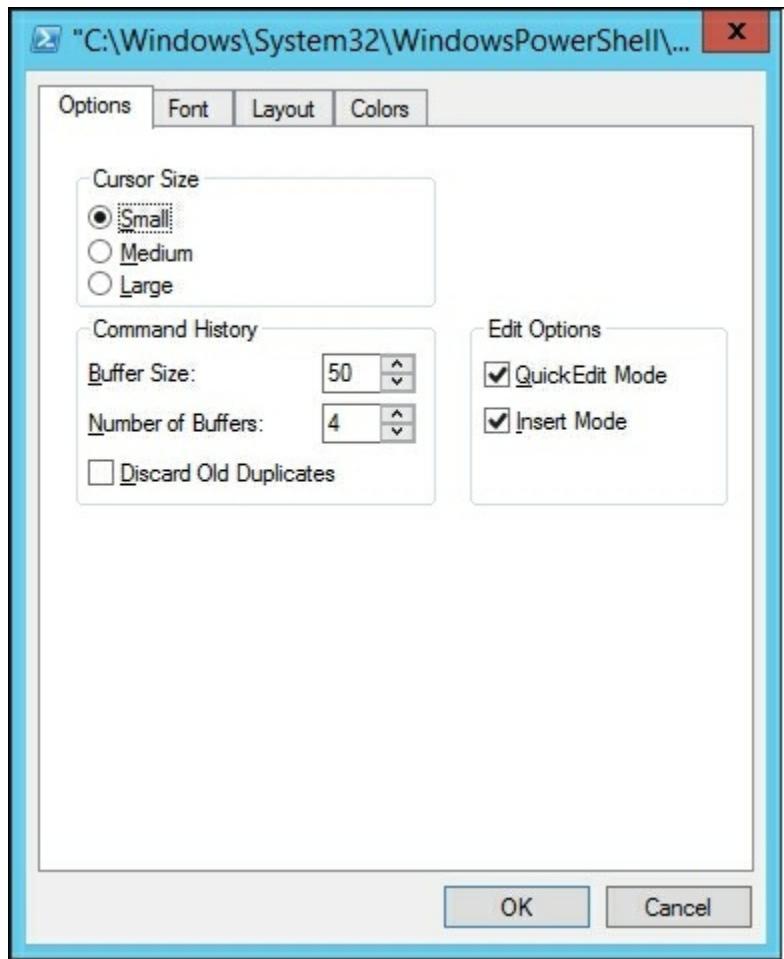
We can change the look and feel of the console host using GUI. To do this, we need to perform the following steps:

Click on the Windows PowerShell icon, which appears in the upper-left corner and select **Properties**. This has four tabs—**Options**, **Font**, **Layout**, and **Colors**. Let's explore each tab.

- **Options**: Using the **Options** tab, we can see **Cursor Size**, **Command History**, and **Edit Options**, and adjust them as we require.

While setting the cursor size, we have three options: **Small**, **Medium**, and **Large**. Here, small is 25 pixels, medium is 50 pixels, and large is 100 pixels.

The following image illustrates the **Options** tab:



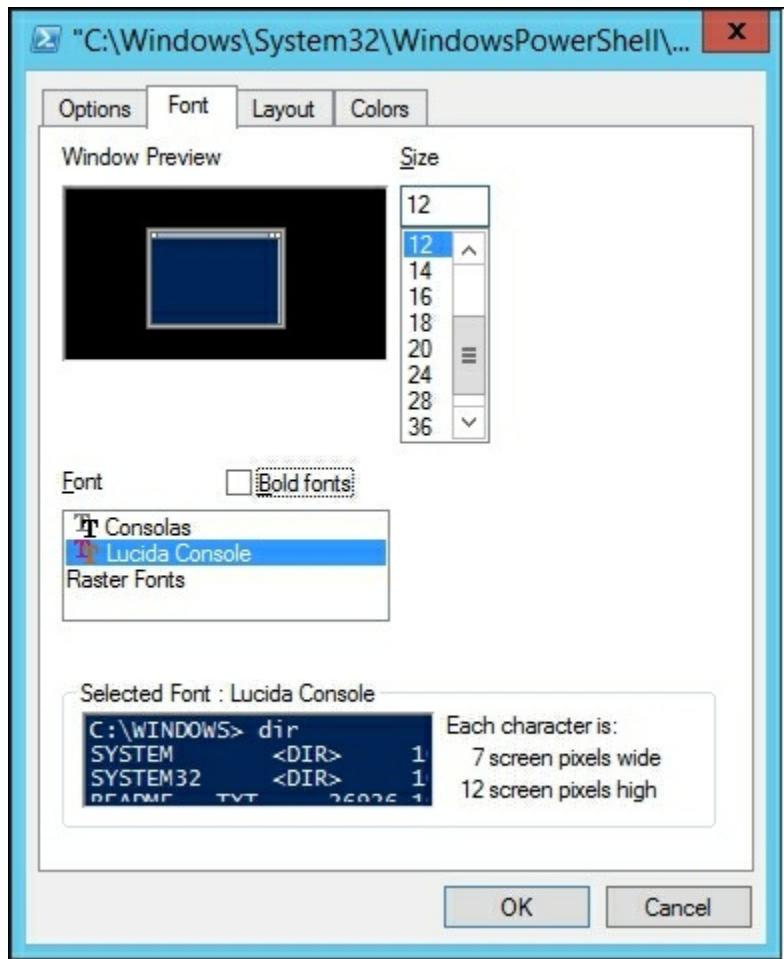
By default, the command history retains the last 50 commands in a buffer. We can customize this by increasing or decreasing the **BufferSize** value.

Tip

\$MaximumHistoryCount is a built-in variable that returns the actual value of the command's history. This takes precedence over GUI.

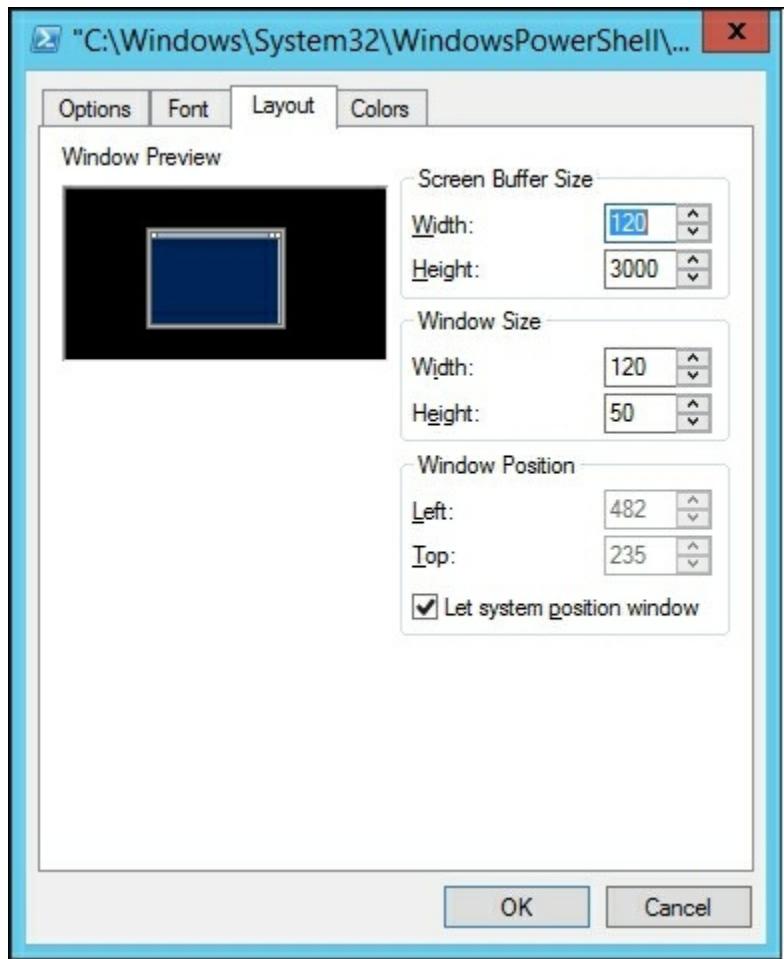
- **Font:** Using the **Font** tab, we can change the size, style, and type. Before applying changes, we can preview the console host using the preview window.

The following image illustrates the **Font** tab:



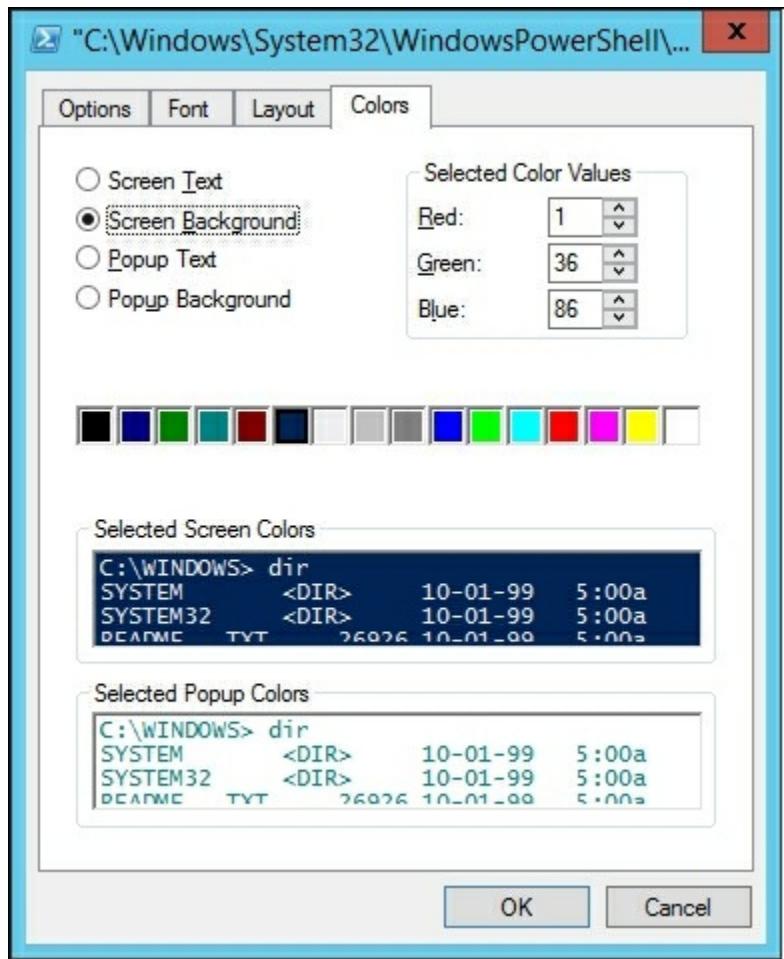
- **Layout:** Using the **Layout** tab, we can set the screen buffer size, window size, and window position. Similarly to the **Font** tab, this also has preview window.

The following image illustrates the **Layout** tab:



- **Colors:** Using the **Colors** tab, we can set the screen text, screen background, pop up text, and popup background values. The current RGB values are shown in the **Selected Color Values** section. The modifications can be previewed before applying.

The following image illustrates the **Colors** tab:



Tip

Press *F7* to see the history; this pops up a tiny window as shown in the following image:

The screenshot shows a Windows PowerShell window titled "Select Administrator: Windows PowerShell". The title bar includes standard window controls (minimize, maximize, close) and a scroll bar on the right.

The main content area displays a list of processes with columns: PID, Thread ID, CPU, CPU %, Process Name, and CPU Time. A portion of the list is highlighted with a yellow background.

```

      855   18   3788   10016 ... 88   0.52   508 lsass
      158   12   2296   6808 ... 93   0.03   2684 msdtc
      440   29   80636   92548 ... 88   3.03   3288 powershell
      214   10   3848   8700 ... 35   0.17   1812 rdpclip
      119    9   2688   2340 ... 12   0.00   1788 rundll32
      198    9   2224   5220 ... 72   0.63   500 services
      55     2   280    1000   4   0.05   268 smss
      431   22   4112   10992 ... 37   0.16   1084 spoolsv
      387   14   3348   9592 ... 94   0.19   564 svchost
      382   34   9532   11908 ... 10   0.20   576 svchost
      327   14   3632   6832 ... 79   0.64   592 svchost
      490   20   11920   15208 ... 17   1.86   720 svchost
      1224  45   14116   29720 ... 85   2.92   768 svchost
      435   24   5080   10828 ... 30   0.22   836 svchost
      572   24   6456   15388 ... 80   0.41   968 svchost
      453   28   9796   14464 ... 20   4.42   1180 svchost
      708   27   68804   81580 ... 49   3.55   1916 svchost
      255   15   2632   7460 ... 98   0.27   1936 svchost
      685    0   108    268    3   4.89   4 System
      156   11   1600   6116 ...
      117    9   1420   5788 ...
      340   37   37216   44520 ...
      566   40   37400   51656 ...
      399   42   45904   56424 ...
      79     8   780    3620 ...
      119    7   1360   5568 ...
      141    7   1204   5000 ...
      249   13   6160   12440 ...

```

A yellow box highlights the command history at the bottom of the window:

```

8: $host.UI.RawUI.CursorSize
9: $host.UI.RawUI.CursorSize = 25
10: Clear-History
11: Get-Service
12: Get-WmiObject -Class *
13: Get-WmiObject -Class Win32_Bios
14: cls
15: get-Service
16: Get-Process
17: Get-Volume

```

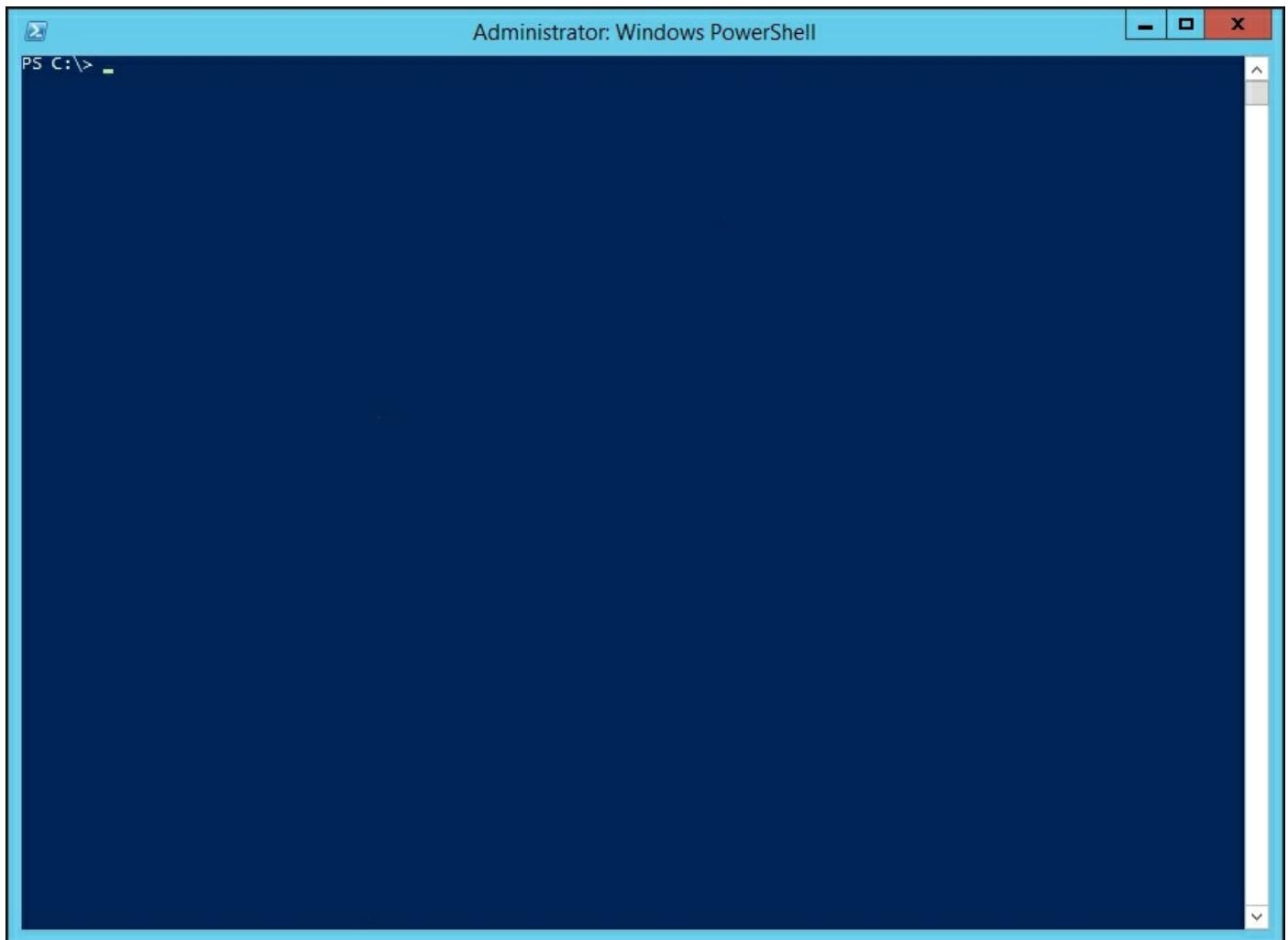
Below the command history, the prompt "PS C:\>" is followed by a horizontal ellipsis (...).

Setting up the console host using PowerShell

The steps we explored under **Graphical User Interface (GUI)** are limited; we can actually do more using PowerShell. Here comes the importance of objects. In this exercise, we will adjust our console settings. To do all this, we need to know is `Get-Host` command-let.

The `Get-Host` cmdlet is used to do more of the Windows PowerShell console setup. In this exercise let's change the title of the PowerShell console host.

The default console host title is shown in the following image:



Administrator: Windows PowerShell is the default title. Now let's change this to Windows PowerShell 5.0 April 2015 Preview. Run the following commands:

```
(Get-Host).UI.RawUI  
(Get-Host).UI.RawUI | GM
```

The preceding code returns the current settings and its members.

The output is illustrated in the following image:

```
Administrator: Windows PowerShell
PS C:\> (Get-Host).UI.RawUI

ForegroundColor      : DarkYellow
BackgroundColor     : DarkMagenta
CursorPosition     : 0,8
WindowPosition      : 0,0
CursorSize          : 50
BufferSize          : 120,3000
WindowSize          : 120,50
MaxWindowSize       : 120,84
MaxPhysicalWindowSize: 274,84
KeyAvailable        : False
WindowTitle         : Administrator: Windows PowerShell

PS C:\> (Get-Host).UI.RawUI | GM

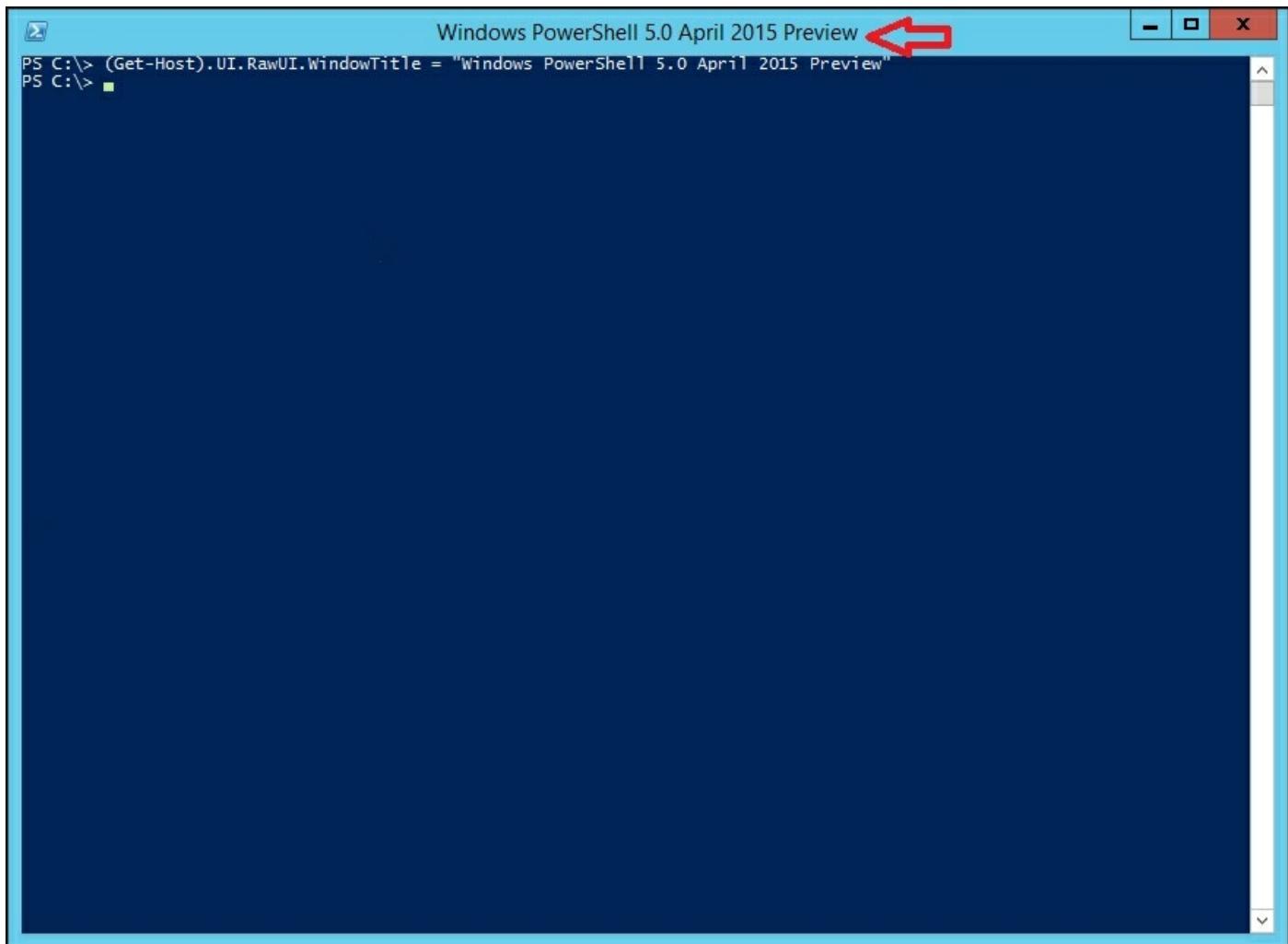
TypeName: System.Management.Automation.Internal.Host.InternalHostRawUserInterface
Name           MemberType Definition
----           -----
Equals         Method   bool Equals(System.Object obj)
FlushInputBuffer Method  void FlushInputBuffer()
GetBufferContents Method System.Management.Automation.Host.BufferCell[,] GetBufferContents(System.Management...
GetHashCode    Method  int GetHashCode()
GetType        Method  type GetType()
LengthInBufferCells Method int LengthInBufferCells(string str, int LengthInBufferCells(string str, int offset...
NewBufferCellArray Method System.Management.Automation.Host.BufferCell[,] NewBufferCellArray(string[] content...
Readkey        Method  System.Management.Automation.Host.KeyInfo ReadKey(System.Management.Automation.Host...
ScrollBufferContents Method void ScrollBufferContents(System.Management.Automation.Host.Rectangle source, Syste...
SetBufferContents Method  void SetBufferContents(System.Management.Automation.Host.Coordinates origin, System...
ToString       Method  string ToString()
BackgroundColor  Property System.ConsoleColor BackgroundColor {get;set;}
BufferSize     Property System.Management.Automation.Host.Size BufferSize {get;set;}
CursorPosition Property System.Management.Automation.Host.Coordinates CursorPosition {get;set;}
CursorSize     Property int CursorSize {get;set;}
ForegroundColor Property System.ConsoleColor ForegroundColor {get;set;}
KeyAvailable   Property bool KeyAvailable {get;}
MaxPhysicalWindowSize Property System.Management.Automation.Host.Size MaxPhysicalWindowSize {get;}
MaxWindowSize  Property System.Management.Automation.Host.Size MaxWindowSize {get;}
WindowPosition  Property System.Management.Automation.Host.Coordinates WindowPosition {get;set;}
WindowSize     Property System.Management.Automation.Host.Size WindowSize {get;set;}
WindowTitle    Property string WindowTitle {get;set;}

PS C:\>
```

WindowTitle is a property for which we can get and set a value and whose datatype is String. We will discuss this in detail in the *Understanding Objects* section. Now, let's execute the following code to change the title of the console host:

```
(Get-Host).UI.RawUI.WindowTitle = "Windows PowerShell 5.0 April  
2015 Preview"
```

The change appears as illustrated in the following image:



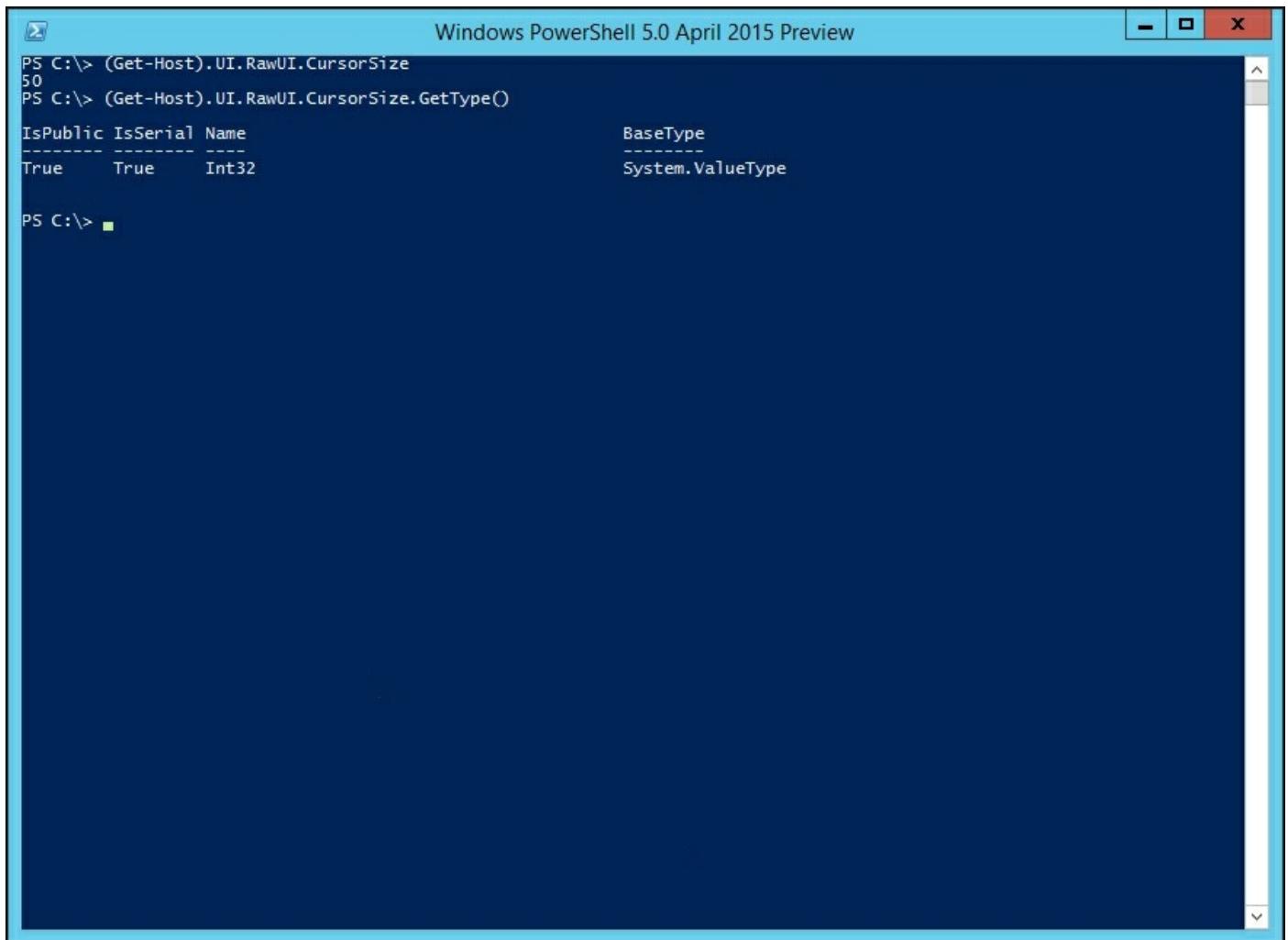
Using GUI, we can set the cursor size to small, medium, or large, which allows only 25, 50, or 100 respectively. But, PowerShell allows us to customize more by executing the following command:

```
(Get-Host).UI.RawUI.CursorSize
```

The previous command returns the current cursor size. Run the following command:

```
(Get-Host).UI.RawUI.CursorSize.GetType()
```

This command returns the type as shown in the following image:



A screenshot of a Windows PowerShell window titled "Windows PowerShell 5.0 April 2015 Preview". The window has a blue header bar with standard window controls (minimize, maximize, close) on the right. The main area of the window shows PowerShell commands and their output:

```
PS C:\> (Get-Host).UI.RawUI.CursorSize
50
PS C:\> (Get-Host).UI.RawUI.CursorSize.GetType()
IsPublic IsSerial Name          BaseType
-----  -----  ---          -----
True    True   Int32         System.ValueType
PS C:\> ■
```

Now, let's change the cursor size to 72 by executing the following command:

```
(Get-Host).UI.RawUI.CursorSize = 72
```

The output is illustrated in the following image:

A screenshot of a Windows PowerShell window titled "Windows PowerShell 5.0 April 2015 Preview". The window has a blue header bar with standard window controls (minimize, maximize, close) on the right. The main area is dark blue. In the top-left corner of the dark area, there is a small red arrow pointing left. The command history at the top shows:

```
PS C:\> (Get-Host).UI.RawUI.CursorSize = 72  
PS C:\> (Get-Host).UI.RawUI.CursorSize  
72  
PS C:\> █ ←
```

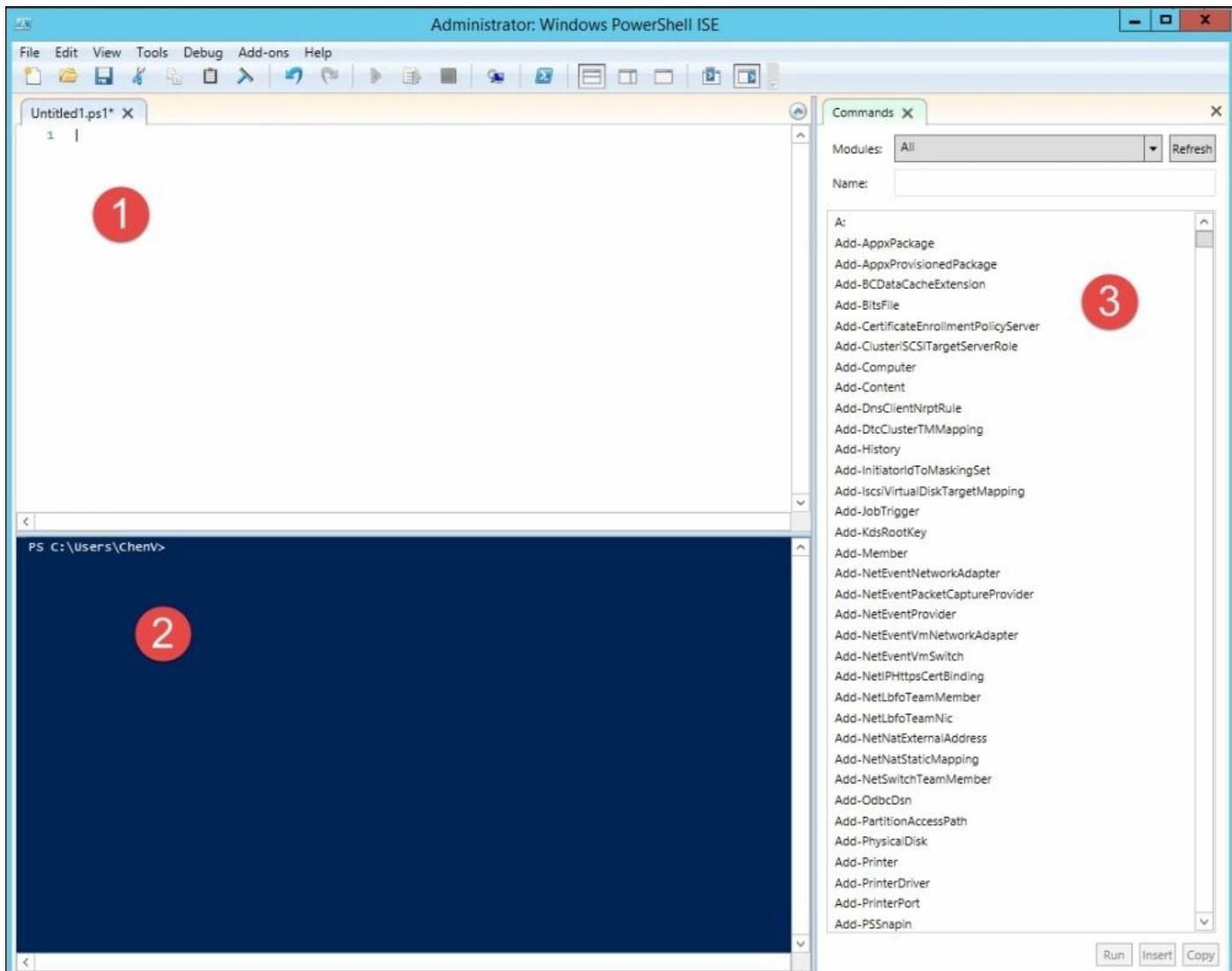
Exploring the Windows PowerShell ISE host using GUI

The Windows PowerShell ISE host is enhanced and exciting in version 5.0. This version has lots of new features and is user friendly. The ISE helps us to write scripts faster than a console. We will explore the features of ISEs in [Chapter 3, Exploring Desired State Configuration](#).

Opening the PowerShell ISE is similar to opening the console host. Click on the start icon, search for `PowerShell`, and open **PowerShell ISE**. The executable file for the PowerShell ISE resides in the same location as console host:

```
%windir%\system32\WindowsPowerShell\v1.0\PowerShell_ISE.exe
```

The following image illustrates the PowerShell ISE:



The PowerShell ISE host is divided into the following sections:

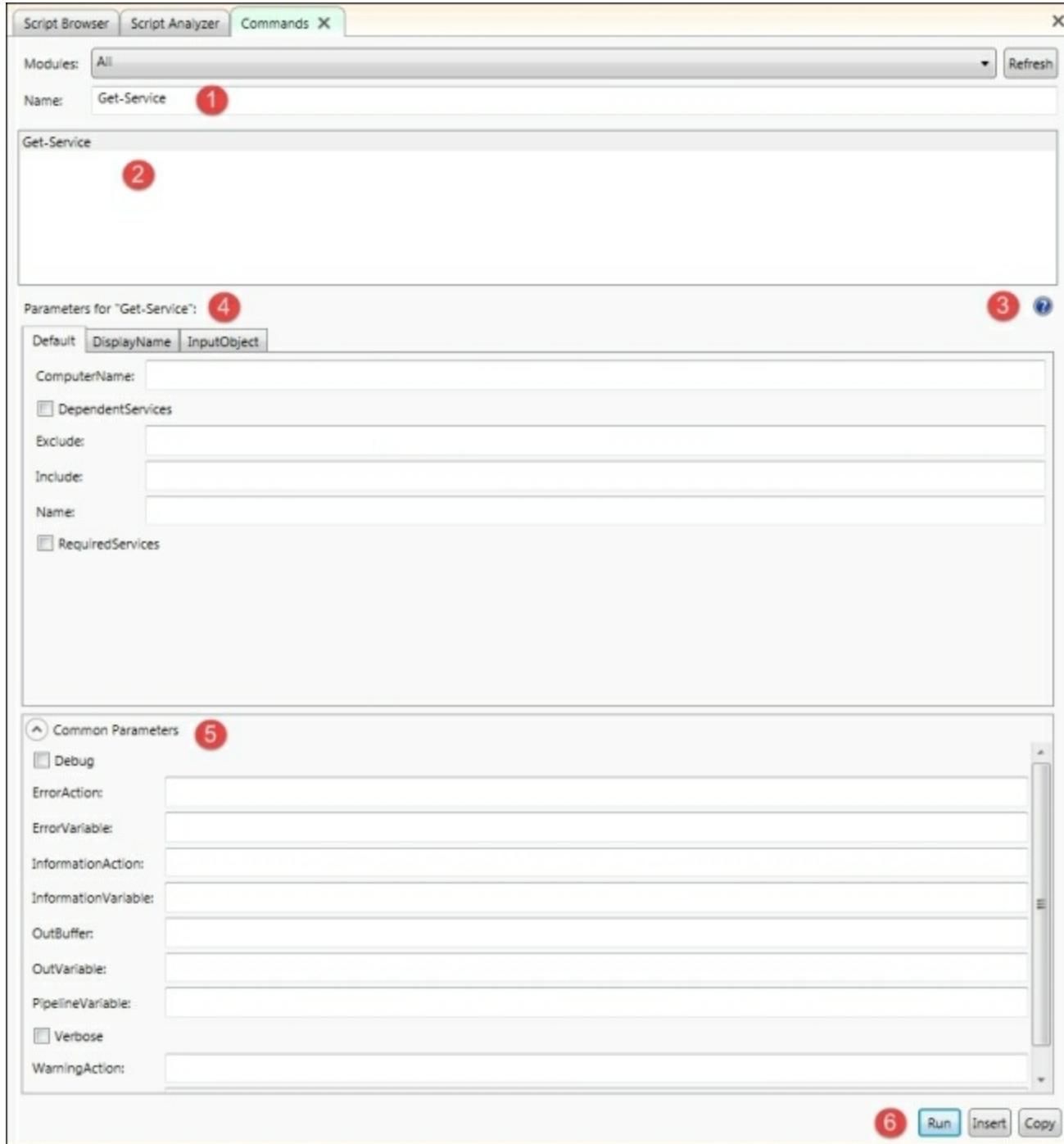
- 1: Script pane
- 2: Command pane
- 3: Commands add-on

The script pane's visibility can be adjusted using the keyboard shortcuts, *Ctrl + 1*, *Ctrl + 2*, *Ctrl + 3*, which sets the script pane to top, right, and maximized, respectively. This makes it easy for us to view the command and result that we need.

The command pane will be hidden if we maximize the script pane.

Let us learn more about commands add-on pane:

1. By default, all the available modules will be shown. Click on the drop-down list to view the modules.
2. Click on **Refresh** if any new modules are loaded.
3. Type the command name, and it will show the output as illustrated in the following image:

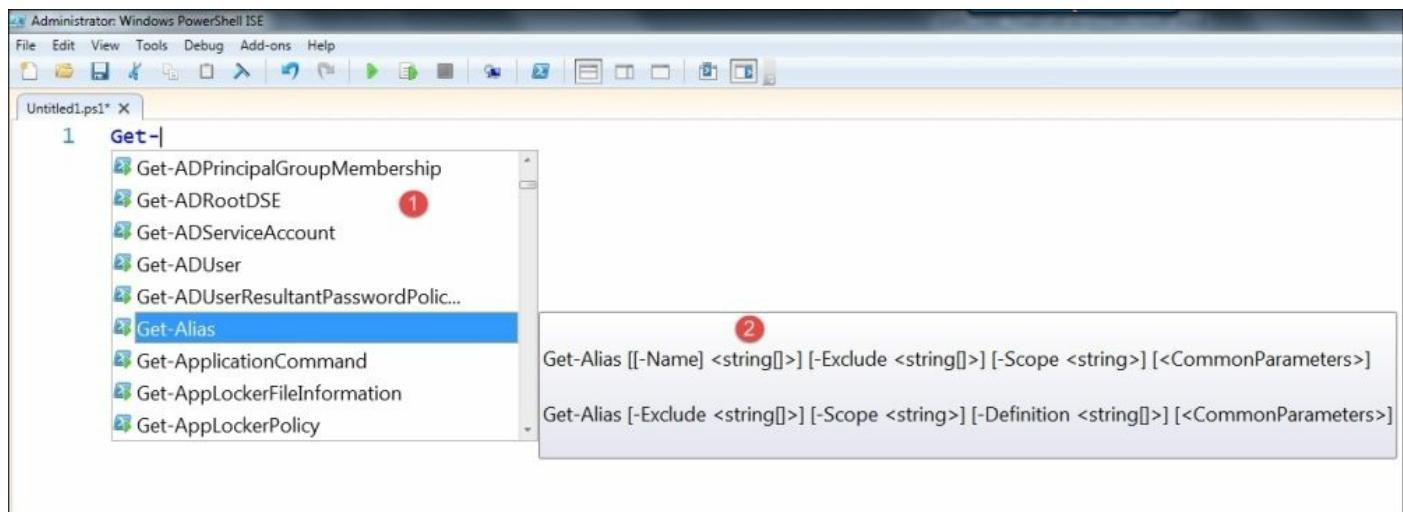


4. Enter the command.
5. Select the command.
6. Click on the help icon, and this opens up the help in GUI.
7. Enter the parameters as required. The * sign denotes mandatory.

8. The common parameters are not always required, but we will use them in the next chapters.
9. We can run the code, copy it, or click on **Insert**, which appears in the command pane. Ensure that you haven't maximized the Script Pane.

An ISE allows us to cut, copy, paste, start IntelliSense, and start the snippets. Similarly to other windows operations, *Ctrl + X*, *Ctrl + C*, and *Ctrl + Y* will cut, copy, and paste, respectively.

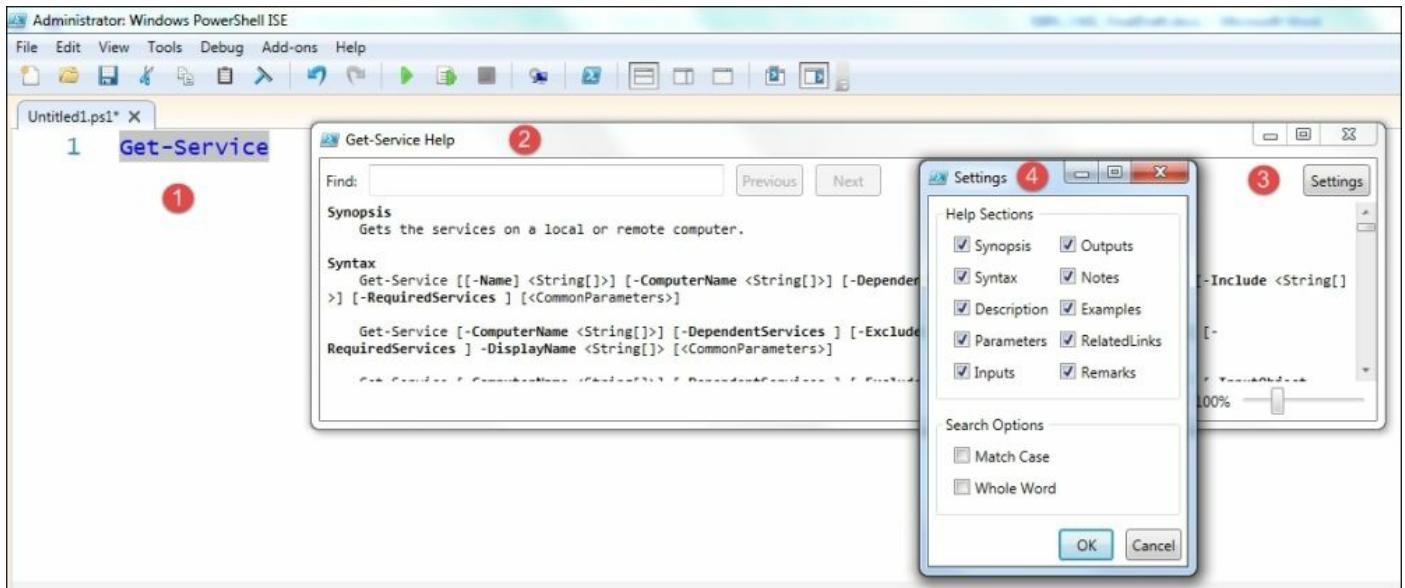
IntelliSense gives us a better discoverability of cmdlets. A bunch of related commands will be shown in the drop-down menu, so we can easily choose the cmdlet we need. The Syntax tooltip lists the parameters to be used with the command, as shown in the following image:



1. As soon as we start typing, IntelliSense shows us the list of commands that begin with the text that is typed. We can scroll down and choose the command we need to execute.
2. The syntax tooltip shows the syntax of the command that we choose.

To get help about the command, simply type the command, select it, and press *F1*.

The output is illustrated in the following image:



The points marked in the figure are explained in the following list:

- 1: Select the command.
- 2: Press *F1*, which brings up the help window.
- 3: Click on **Settings**.
- 4: Select the sections you need to know. For example, unselect **Examples** to hide the example section in the help window.

Benefits of an ISE

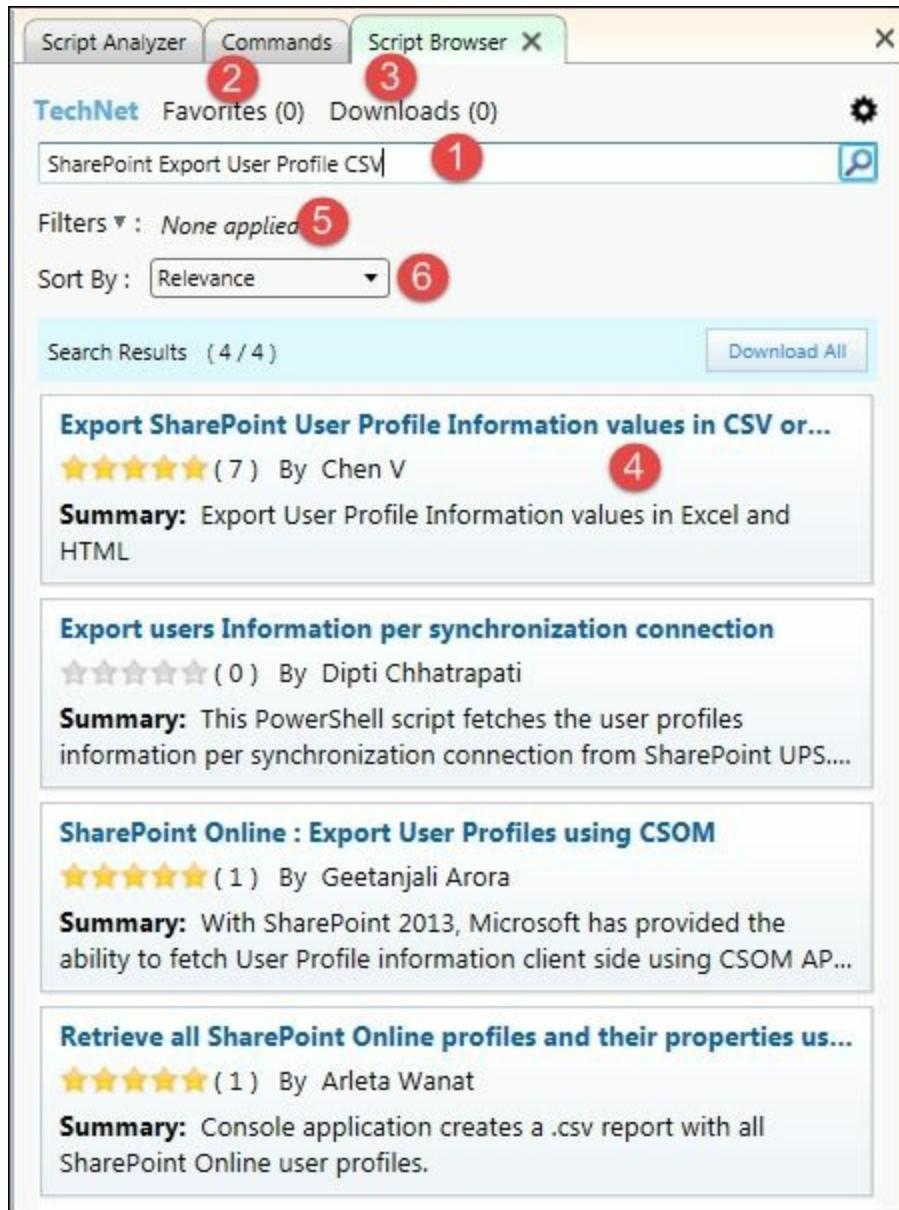
In comparison with the PowerShell console host, an ISE has a lot of benefits, and I will list a few of them here:

- It's easy to use
- It's very user friendly
- There are shortcut keys to manage the script
- Writing the DSC configuration code is easy
- The snippets make our job much easier
- There is a script browser
- We have the Command add-on
- It shows a squiggly line in case of errors

The PowerShell ISE script browser

PowerShell ISE has a script browser feature that allows us to search for the script in TechNet's script gallery. This is optional but worth using! This is because using **Script Browser**, we can download and use the scripts available in TechNet's script gallery.

The following image illustrates Script Browser:

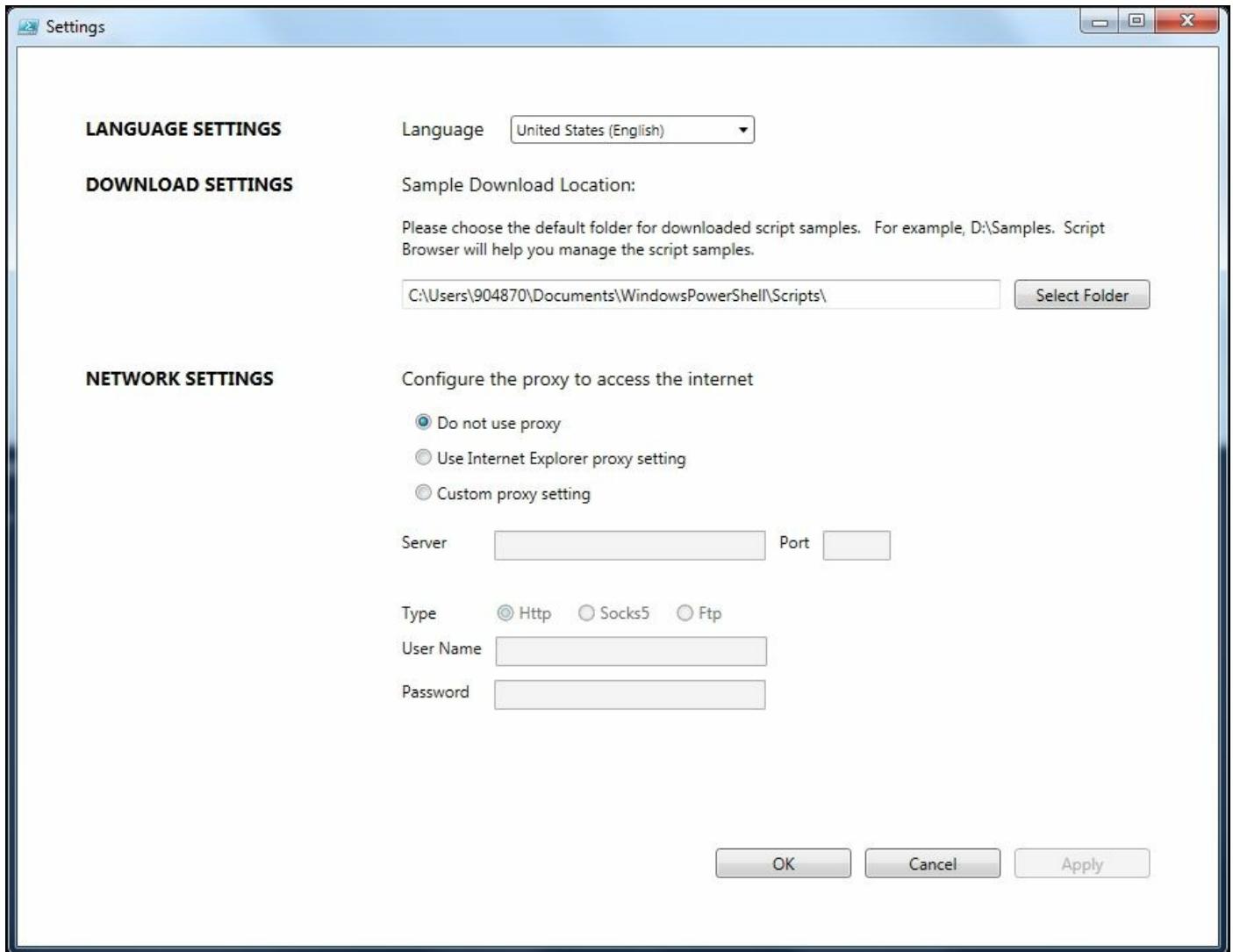


Following are a few options in **Script Browser**:

- You can search for the script you need. Type the script's name in the textbox provided, right-click on the script, and select either **Add to favorites** or the **Download** option.

- Your favorite scripts appears in the **Favorites** tab.
- The scripts that you download will appear in the **Downloads** tab.
- You can set the language, download locations and network settings by clicking on the settings icon in the upper-right corner.

The following image shows the **Settings** window of **Script Browser**:



Some benefits of Script Browser are as follows:

- Script Browser allows us to filter the search based on technologies
- Script Browser allows us to sort the script by popularity, ratings, and so on

Note

The link to download Script Browser is <http://www.microsoft.com/en->

[us/download/details.aspx?id=42525.](#)

Using an interactive shell

Windows PowerShell can be used either interactively or as a script automation. Before we take a look at script automations, let's explore the use of an interactive shell.

We can do arithmetic operations in Windows PowerShell as follows:

```
2 + 2
```

This will return 4. Consider the following operation:

```
2 + 2 * 3
```

This will return 8.

Windows PowerShell does left-to-right arithmetic operations according to the precedence rules. In the preceding expressions, PowerShell performed the operation as $2 * 3 + 2$. We need to use parenthesis, as with $(2 + 2) * 3$, which return the result as 12. Here, $(2 + 2)$ is known as grouping.

Using Windows PowerShell, we can do much more, such as system unit calculations, hexadecimal conversions, and so on, and few examples are as follows:

```
4GB / 1MB
```

This returns 4096.

```
(0xC85).ToChar($_)
```

This returns the following character:

```
'¤'
```

In Windows PowerShell, we can use .NET classes. Here's an example:

```
[System.Math]::PI
```

This returns the value of pi, which is 3.14159265358979.

Windows PowerShell allows us to use class directly, without using the namespace, shown as follows:

```
[Math]::PI
```

Note

Refer to the following MSDN link for the `System.Math` class:

<https://msdn.microsoft.com/en-us/library/system.math%28v=vs.110%29.aspx>

The command shown before was just an example. With reference to this, we can build code as per our needs using .NET classes.

Windows PowerShell cmdlets

In this topic, we will cover the following:

- Exploring Windows PowerShell commands
- Exploring Windows PowerShell modules
- Getting help
- Understanding aliases, expressions, objects, pipelines, filtering, and formatting
- Scripting with Windows PowerShell

Windows PowerShell is designed to execute four kinds of named commands:

- Cmdlets: These are .NET programs designed to interact with PowerShell
- Scripts: These are files with the `FileName.ps1` extension
- Functions: These are a block of code that are very helpful for script organization
- Native window commands: These are internal commands such as `MKDIR`, `CHDIR`, and so on

The internal commands of Windows PowerShell are called cmdlets.

Cmdlets are always named in the verb-and-noun combination. This appears as *Verb-Noun*, for example Get-Service and Stop-Service, where Get and Stop are verbs, and Service is a noun.

(Verb)—Action

(Noun)—Something to be acted on

For example, Get-Service

Let's execute the following command:

Get-Command

The output of this command is illustrated in the following image:

CommandType	Name	Version	Source
Alias	Add-WAPackEnvironment	0.8.7	Azure
Alias	Begin-WebCommitDelay	1.0.0.0	WebAdministration
Alias	Disable-WAPackWebsiteApplicationDiagnostic	0.8.7	Azure
Alias	Enable-WAPackWebsiteApplicationDiagnostic	0.8.7	Azure
Alias	End-WebCommitDelay	1.0.0.0	WebAdministration
Alias	Fix-It	1.2.0.132	pspx
Alias	Get-AzureStorageContainerAcl	0.8.7	Azure
Alias	Get-WAPackEnvironment	0.8.7	Azure
Alias	Get-WAPackPublishSettingsFile	0.8.7	Azure
Alias	Get-WAPackSBLocation	0.8.7	Azure
Alias	Get-WAPackSBNamespace	0.8.7	Azure
Alias	Get-WAPackSubscription	0.8.7	Azure
Alias	Get-WAPackWebsite	0.8.7	Azure
Alias	Get-WAPackWebsiteDeployment	0.8.7	Azure
Alias	Get-WAPackWebsiteLocation	0.8.7	Azure
Alias	Get-WAPackWebsiteLog	0.8.7	Azure
Alias	Import-WAPackPublishSettingsFile	0.8.7	Azure
Alias	Invoke-Hive	0.8.7	Azure
Alias	New-WAPackSBNamespace	0.8.7	Azure
Alias	New-WAPackWebsite	0.8.7	Azure
Alias	Remove-WAPackEnvironment	0.8.7	Azure
Alias	Remove-WAPackSBNamespace	0.8.7	Azure
Alias	Remove-WAPackSubscription	0.8.7	Azure
Alias	Remove-WAPackWebsite	0.8.7	Azure
Alias	Restart-WAPackWebsite	0.8.7	Azure
Alias	Restore-WAPackWebsiteDeployment	0.8.7	Azure
Alias	Save-WAPackWebsiteLog	0.8.7	Azure
Alias	Select-WAPackSubscription	0.8.7	Azure
Alias	Set-WAPackEnvironment	0.8.7	Azure
Alias	Set-WAPackSubscription	0.8.7	Azure
Alias	Set-WAPackWebsite	0.8.7	Azure
Alias	Show-WAPackPortal	0.8.7	Azure
Alias	Show-WAPackWebsite	0.8.7	Azure
Alias	Start-CopyAzureStorageBlob	0.8.7	Azure

The previous command will list all the installed commands from the computer and retrieve only the cmdlets, functions, and aliases.

To retrieve only cmdlets, we can use the `CommandType` parameter, as shown in the following command:

```
Get-Command - CommandType Cmdlet
```

Note

Windows PowerShell supports tab completion. For example, let's run the following command:

```
Get-Comm + Tab + -Co + Tab + c + Tab
```

When we type out this command, it will result as follows:

```
Get-Command - CommandType Cmdlet
```

Now, let's explore the commands. In the following example, we will take a look at the cmdlet `Get-Service`. Since we know the cmdlet up front, let's collect more information such as the cmdlet version and source, as follows:

```
Get-Command Get-Service
```

The output of the preceding command is illustrated in the following image:

CommandType	Name	Version	Source
Cmdlet	Get-Service	3.1.0.0	Microsoft.PowerShell.Management

The points marked in the figure are explained in the following list:

- 1: This is the type of the command
- 2: This is the name of the command
- 3: This indicates the version (From Windows PowerShell 5.0 onward)
- 4: This indicates the module's name

To retrieve the commands for a specific module, we need to use the `Module` parameter, which accepts the module name (**Source**) as we saw in bullet 4. The following is an example of the command:

```
Get-Command -Module Microsoft.PowerShell.Management
```

This outputs all the commands available in the `Microsoft.PowerShell.Management` module.

The `Get-Service` command will retrieve all the services running on your local computer.

Note

Refer to the following link to get more information on the `ServiceController` class:

<https://msdn.microsoft.com/en-us/library/system.serviceprocess.servicecontroller%28v=vs.110%29.aspx>

Let's see the alternate of the `Get-Service` cmdlet using the .NET class. To do this, let's find the `TypeName` of the `Get-Service` cmdlet by executing `Get-Service | Get-Member` and this gives the `TypeName` as `System.ServiceProcess.ServiceController`. Windows PowerShell allows us to use this directly as follows:

```
[System.ServiceProcess.ServiceController]::GetServices()
```

Now, let's take a look at how the command that we just discussed works.

Here, we will consume the .NET `ServiceController` class in PowerShell and invoke the `GetServices()` method.

The `GetServices` method has an overload with a `machineName` parameter. To find this, we will execute the following command:

```
[System.ServiceProcess.ServiceController]::GetServices
```

The difference is that here, we did not use parentheses. The output of

`OverloadDefinitions` is as follows:

OverloadDefinitions

```
static System.ServiceProcess.ServiceController[] GetServices()  
  
static System.ServiceProcess.ServiceController[]  
GetServices(string machineName)
```

We can query the service information of a remote computer by passing the host name of the remote computer as the `machineName` parameter, as follows:

```
[System.ServiceProcess.ServiceController]::GetServices('Remote  
Server')
```

Getting help

This is the most important and interesting topic. No matter what technology we consume, all we need to know is the way to get help for it. Most IT professionals and developers say that they use Google to find this.

For PowerShell, help is much more focused. It's very difficult to remember all the commands. So, we can search for a command and find help for the same. Let's take a look at how to seek help for Windows PowerShell cmdlets.

```
Get-Help Get-Service
```

The output is illustrated in the following image:

The screenshot shows a Windows PowerShell window with the following command and its help output:

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

NAME ①
Get-Service

SYNOPSIS ②
Gets the services on a local or remote computer.

SYNTAX ③
Get-Service [*-Name* <String[]>] [*-ComputerName* <String[]>] [*-DependentServices*] [*-Exclude* <String[]>] [*-Include* <String[]>] [*-RequiredServices*] [*<CommonParameters>*]
Get-Service [*-ComputerName* <String[]>] [*-DependentServices*] [*-Exclude* <String[]>] [*-Include* <String[]>] [*-RequiredServices*] [*-DisplayName* <String[]>] [*<CommonParameters>*]
Get-Service [*-ComputerName* <String[]>] [*-DependentServices*] [*-Exclude* <String[]>] [*-Include* <String[]>] [*-InputObject* <ServiceController[]>] [*-RequiredServices*] [*<CommonParameters>*]

DESCRIPTION ④
The Get-Service cmdlet gets objects that represent the services on a local computer or on a remote computer, including running and stopped services.
You can direct Get-Service to get only particular services by specifying the service name or display name of the services, or you can pipe service objects to Get-Service.

RELATED LINKS ⑤
Online version: <http://go.microsoft.com/fwlink/p/?linkid=298583>
New-Service
Restart-Service
Resume-Service
Set-Service
Start-Service
Stop-Service
Suspend-Service

REMARKS ⑥
To see the examples, type: "get-help Get-Service -examples".
For more information, type: "get-help Get-Service -detailed".
For technical information, type: "get-help Get-Service -full".
For online help, type: "get-help Get-Service -online"

The sections in the image are explained as follows:

- **NAME:** This is the name of the command (`Get-Service`)
- **SYNOPSIS:** This is the abstract of the command
- **SYNTAX:** This gives us the syntax of the commands, which includes all its parameters and its type
- **DESCRIPTION:** This is the description of the command whose help we are looking for
- **RELATED LINKS:** This contains the URL of online versions of the command, and other commands related to the one we are looking for help regarding
- **REMARKS:** This will guide us to explore examples, detailed information, full help, and online help

If more information than that fitting the page view is to be displayed, the console is paginated. For example, if we execute the `Get-Help Get-Service -Detailed | more` command, the details will output as shown in the following image:

The screenshot shows an Administrator Windows PowerShell window with the title bar "Administrator: Windows PowerShell". The content displays the help documentation for the Get-Service cmdlet. The help text includes sections for NAME, SYNTAX, DESCRIPTION, PARAMETERS, EXAMPLES, and EXAMPLE 1. It provides detailed information about each parameter, such as -ComputerName, -DependentServices, -DisplayName, -InputObject, -Name, -RequiredServices, and -CommonParameters. The output ends with examples of how to use the cmdlet.

```

NAME
    Get-Service
SYNOPSIS
    Gets the services on a local or remote computer.
SYNTAX
    Get-Service [[-Name <String[]>] [-ComputerName <String[]>] [-DependentServices] [-Exclude <String[]>] [-Include <String[]>] [-RequiredServices] [<CommonParameters>]
    Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude <String[]>] [-Include <String[]>] [-RequiredServices] [-DisplayName <String[]>] [<CommonParameters>]
    Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude <String[]>] [-Include <String[]>] [-InputObject <ServiceController[]>] [-RequiredServices] [<CommonParameters>]
DESCRIPTION
    The Get-Service cmdlet gets objects that represent the services on a local computer or on a remote computer, including running and stopped services.
    You can direct Get-Service to get only particular services by specifying the service name or display name of the services, or you can pipe service objects to Get-Service.
PARAMETERS
    -ComputerName <String[]>
        Gets the services running on the specified computers. The default is the local computer.
        Type the NetBIOS name, an IP address, or a fully qualified domain name of a remote computer. To specify the local computer, type the computer name, a dot <.>, or "localhost".
        This parameter does not rely on Windows PowerShell remoting. You can use the ComputerName parameter of Get-Service even if your computer is not configured to run remote commands.
    -DependentServices [SwitchParameter]
        Gets only the services that depend upon the specified service.
        By default, Get-Service gets all services.
    -DisplayNames <String[]>
        Specifies the display names of services to be retrieved. Wildcards are permitted. By default, Get-Service gets all services on the computer.
    -Exclude <String[]>
        Omits the specified services. The value of this parameter qualifies the Name parameter. Enter a name element or pattern, such as "&m". Wildcards are permitted.
    -Include <String[]>
        Retrieves only the specified services. The value of this parameter qualifies the Name parameter. Enter a name element or pattern, such as "&m". Wildcards are permitted.
    -InputObject <ServiceController[]>
        Specifies ServiceController objects representing the services to be retrieved. Enter a variable that contains the objects, or type a command or expression that gets the objects. You can also pipe a service object to Get-Service.
    -Name <String[]>
        Specifies the service names of services to be retrieved. Wildcards are permitted. By default, Get-Service gets all of the services on the computer.
    -RequiredServices [SwitchParameter]
        Gets only the services that this service requires.
        This parameter gets the value of the ServicesDependedOn property of the service. By default, Get-Service gets all services.
<CommonParameters>
    This cmdlet supports the common parameters: -Verbose, -Debug,
    -ErrorAction, -ErrorVariable, -OutBuffer, -PipelineVariable, and -OutVariable. For more information, see
    about_CommonParameters (http://go.microsoft.com/fwlink/?LinkId=113216).
EXAMPLE 1
PS C:\>get-service
This command retrieves all of the services on the system. It behaves as though you typed "get-service *". The default display shows the status, service name, and display name of each service.

----- EXAMPLE 2 -----
PS C:\>get-service wmi
This command retrieves services with service names that begin with "WMI" (the acronym for Windows Management Instrumentation).

    Paginated
    More -- EXAMPLE 3

```

If we press *Enter*, we can view one line after another, whereas pressing the space key will give a page view.

Note

Keep your files updated using the Update-Help cmdlet as shown in the following command. Ensure that your machine has internet connectivity and execute the following command:

Update-Help -Verbose

This cmdlet is designed to download and install help files on our computer.

The output is illustrated in the following image:

```
[Administrator: Windows PowerShell
PS C:\> Update-Help -Verbose
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?linkid=398785"
Updating Help for module Microsoft.PowerShell.ODataUtils
Locating Help Content...
VERBOSE: Performing the operation 'Update-Help' on target 'Microsoft.PowerShell.Utility'. Current Version: 5.0.1.0, Available Version: 5.0.1.0, UICulture: en-US".
VERBOSE: Microsoft.PowerShell.Utility: The most current Help files are already installed.. Culture en-US Version 5.0.1.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?linkid=398785"
VERBOSE: Performing the operation 'Update-Help' on target 'CimCmdlets'. Current Version: 5.0.0.0, Available Version: 5.0.0.0, UICulture: en-US".
VERBOSE: CimCmdlets: The most current Help files are already installed.. Culture en-US Version 5.0.0.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?linkid=398785"
VERBOSE: Performing the operation 'Update-Help' on target 'ISE'. Current Version: 5.0.1.0, Available Version: 5.0.1.0, UICulture: en-US".
VERBOSE: ISE: The most current Help files are already installed.. Culture en-US Version 5.0.1.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?linkid=398785"
VERBOSE: Your connection has been redirected to the following URI: "http://download.microsoft.com/download/1/3/2/137FFD122-B0E8-4B9A-B0B8-445011526028/"
VERBOSE: Performing the operation 'Update-Help' on target 'Microsoft.PowerShell.Archive'. Current Version: 5.0.0.0, Available Version: 5.0.0.0, UICulture: en-US".
VERBOSE: Microsoft.PowerShell.Archive: The most current Help files are already installed.. Culture en-US Version 5.0.0.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?linkid=398785"
VERBOSE: Your connection has been redirected to the following URI: "http://download.microsoft.com/download/E/5/4/E54862BD-FD65-4C76-91E4-3C1958EC1694/"
VERBOSE: Performing the operation 'Update-Help' on target 'Microsoft.PowerShell.Diagnostics'. Current Version: 5.0.1.0, Available Version: 5.0.1.0, UICulture: en-US".
VERBOSE: Microsoft.PowerShell.Diagnostics: The most current Help files are already installed.. Culture en-US Version 5.0.1.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?linkid=398785"
VERBOSE: Your connection has been redirected to the following URI: "http://download.microsoft.com/download/9/3/0/938BD975-BD03-410F-B33-9422DF59E30/"
VERBOSE: Performing the operation 'Update-Help' on target 'Microsoft.PowerShell.Host'. Current Version: 5.0.1.0, Available Version: 5.0.1.0, UICulture: en-US".
VERBOSE: Microsoft.PowerShell.Host: The most current Help files are already installed.. Culture en-US Version 5.0.1.0
VERBOSE: Resolving URI: "http://go.microsoft.com/fwlink/?linkid=398785"
VERBOSE: Your connection has been redirected to the following URI: "http://download.microsoft.com/download/C/9/3/C93133B9-5D2B-4EAA-AE09-EFEDB664545"
VERBOSE: Your connection has been redirected to the following URI: "http://ak.or.download.microsoft.com/download/C/9/3/C93133B9-5D2B-4EAA-AE09-EFEDB664545/"
VERBOSE: Performing the operation 'Update-Help' on target 'Microsoft.PowerShell.ODataUtils'. Current Version: 5.0.1.0, Available Version: 5.0.1.0, UICulture: en-US".
VERBOSE: Microsoft.PowerShell.ODataUtils: The most current Help files are already installed.. Culture en-US Version 5.0.1.0
```

Ensure that you have an Internet connection while updating your help. The reason for updating help is to keep the help document up-to-date. Let us learn more about the `Get-Help` cmdlet:

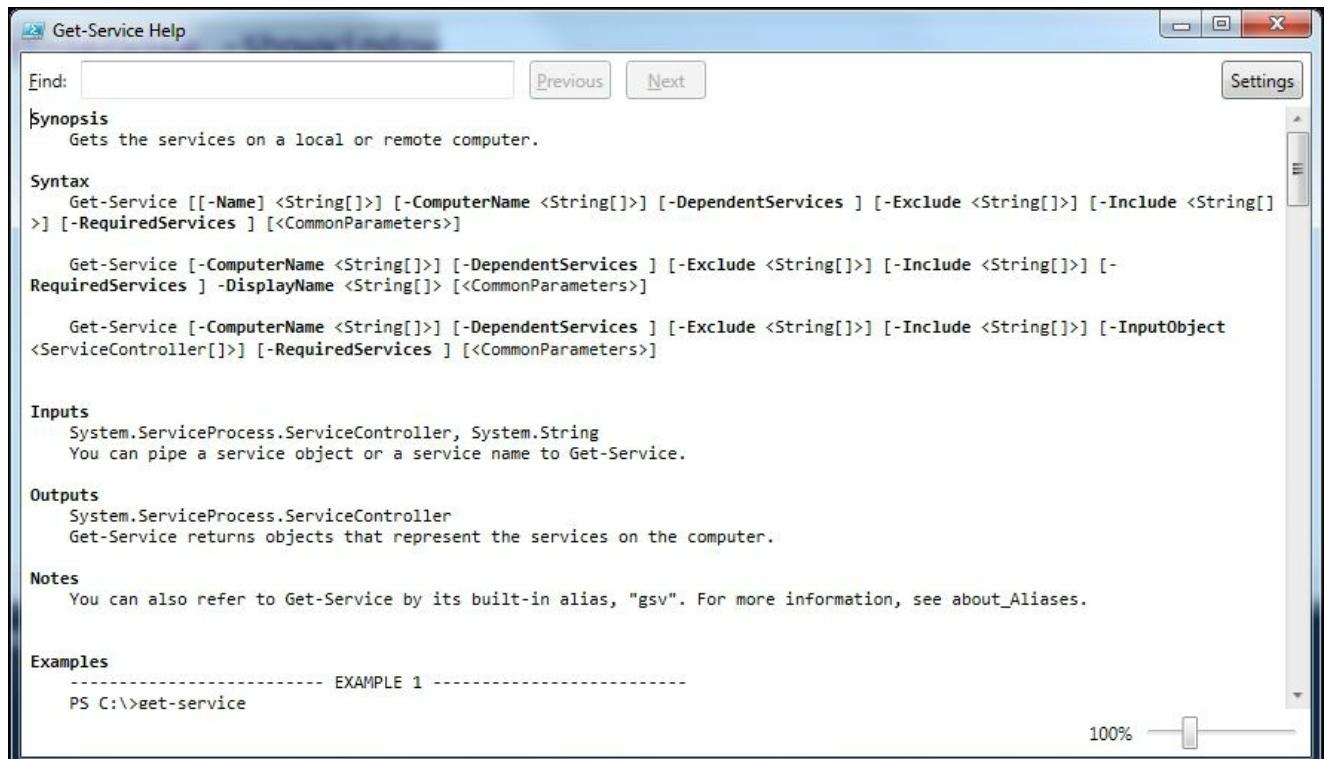
- The `Help` cmdlet is an alias for `Get-Help` (Aliases will be covered in the next section).
- The `Get-Help` cmdlet allows us to view the online help using the `Online` parameter. The following command will open the online URL in the default web browser:

Get-Help Get-Service -Online

- The `Get-Help` cmdlet allows us to view the help content in a separate user interface. The following is the command that needs to be executed:

Get-Help Get-Service -ShowWindow

The output of the preceding code is illustrated in the following image:



- To view only syntax, we can execute the following code:

```
(Get-Help Get-Service).Syntax
```

Understanding aliases

Using aliases in Windows PowerShell is not advised by many. The reason for this is readability and understandability. Developers are comfortable using an alias because it's easier, but the difficulty is in remembering the alias for each cmdlet.

The bottom line is that aliases are shortcuts for Windows PowerShell cmdlets.

Windows PowerShell has two types of aliases:

- The built-in alias: This is an alias that represents PowerShell's native cmdlets
- The user-defined alias: This is an alias created by us for specific needs.

The following command retrieves all the commands available in the

Microsoft.PowerShell.Management module:

Get-Command -Module Microsoft.PowerShell.Management

The following image shows the output of the previous command:

CommandType	Name	Version	Source
Cmdlet	Add-Computer	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Add-Content	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Checkpoint-Computer	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Clear-Content	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Clear-EventLog	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Clear-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Clear-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Clear-RecycleBin	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Complete-Transaction	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Convert-Path	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Copy-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Copy-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Debug-Process	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Disable-ComputerRestore	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Enable-ComputerRestore	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-ChildItem	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Clipboard	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-ComputerRestorePoint	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Content	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-ControlPanelItem	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-EventLog	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-HotFix	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-ItemPropertyValue	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Location	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Process	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-PSDrive	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-PSPrinter	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Service	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Transaction	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-WmiObject	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Invoke-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Invoke-WmiMethod	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Join-Path	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Limit-EventLog	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Move-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Move-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management

Using the following alias, we can achieve the same output as with `-like`:

gcm -Module Microsoft.PowerShell.Management

Here, `gcm` is an alias or shortcut for the `Get-Command` cmdlet.

Let's explore all the commands related to an alias. The module name for aliases' commands is `Microsoft.PowerShell.Utility`:

Get-Command -Name '*Alias*'

The output of this command is illustrated in the following image:

```

Select Administrator: Windows PowerShell
PS C:\> Get-Command -Name '*Alias*'

 CommandType      Name          Version   Source
-----          ----          -----   -----
 Function        Set-PhElementAliases  1.2.3     PoshBoard
 Cmdlet          Export-Alias    3.1.0.0   Microsoft.PowerShell.Utility
 Cmdlet          Get-Alias      3.1.0.0   Microsoft.PowerShell.Utility
 Cmdlet          Import-Alias   3.1.0.0   Microsoft.PowerShell.Utility
 Cmdlet          New-Alias     3.1.0.0   Microsoft.PowerShell.Utility
 Cmdlet          Set-Alias     3.1.0.0   Microsoft.PowerShell.Utility

PS C:\>

```

For now, ignore the `PoshBoard` module; we will discuss modules in [Chapter 2, Unleashing Development Skills Using Windows PowerShell 5.0.](#)

To find an alias of any given command, we can simply execute the following code:

```
Get-Alias -Definition Get-Alias
```

The output of this command is as follows:

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

CommandType	Name
Version	Source
-----	-----
-----	-----
Alias	gal -> Get-Alias

Here, `gal` is an alias of the `Get-Alias` cmdlet. The `Get-Alias` cmdlet will retrieve all the available aliases in your local computer.

PowerShell allows us to create aliases for any given valid cmdlet. To create a new alias, we execute the following command:

```
New-Alias -Name W -Value Get-WmiObject -Description "Learning PowerShell" -Verbose
```

The output of the command we just discussed is as follows:

```
VERBOSE: Performing the operation "New Alias" on target "Name: W Value: Get-WmiObject"
```

Now, let's take a look at how the command we just discussed works.

The `New-Alias` command is used to create an alias.

The `w` part is a friendly name. You can choose any name you need, but you can't create a new alias with the existing, used names.

The `Value` command is used here. We also used the `Get-WmiObject` command, and that's our definition.

The `Description` parameter is for our reference.

The `Verbose` parameter just shows the action performed by the message.

The output of the preceding command is illustrated in the following image:

```
PS C:\windows\system32> Get-Alias -Name w | Select *
```

HelpUri	:	http://go.microsoft.com/fwlink/p/?linkid=290505
ResolvedCommandName	:	Get-WmiObject
DisplayName	:	W -> Get-WmiObject
ReferencedCommand	:	Get-WmiObject
ResolvedCommand	:	Get-WmiObject
Definition	:	Get-WmiObject②
Options	:	None
Description	:	Learning PowerShell③
OutputType	:	{}
Name	:	W ①
CommandType	:	Alias ④
Source	:	
Version	:	
Visibility	:	Public ⑤
ModuleName	:	
Module	:	
RemotingCapability	:	OwnedByCommand
Parameters	:	{[Class, System.Management.Automation.ParameterMetadata], [Recurse, System.Management.Automation.ParameterMetadata], [Property, System.Management.Automation.ParameterMetadata], [Filter, System.Management.Automation.ParameterMetadata]...}
ParameterSets	:	

The sections in the image are explained as follows:

- **Name:** This is the name of the alias.
- **Definition:** Here, this is the `Get-WmiObject` command. It may be any command.

- **Description:** This is the description of the alias.
- **CommandType:** Here, this is Alias.
- **Visibility:** Here, this is Public.

Using the `Set-Alias` command, we can do the same, but it allows us to change the association later. In the preceding example, we created an alias named `w` for the `Get-WmiObject` command. If we try to create an alias with the same name for another command, the following error appears:

```
PS C:\> New-Alias -Name w -Value Get-Service -Description Learning
New-Alias : The alias is not allowed, because an alias with the name 'w' already exists.
At line:1 char:1
+ New-Alias -Name w -Value Get-Service -Description Learning
+ ~~~~~
+ CategoryInfo          : ResourceExists: (w:string) [New-Alias], SessionStateException
+ FullyQualifiedErrorId : AliasAlreadyExists,Microsoft.PowerShell.Commands.NewAliasCommand
```



In this scenario, we can use the `Set-Alias` command to change the alias' association. Run the following code:

```
PS C:\> Set-Alias -Name w -Value Get-Service -Description
'Change Association' -Verbose
VERBOSE: Performing the operation "Set Alias" on target "Name:
w Value: Get-Service"
```

Note

Best practice to use an alias is when you are controlling the environment entirely.

Ensure that you have made a note about alias in your script. This helps others to understand and troubleshoot in case of any failures.

If you create more aliases in your machine, you can move them to other machines using the following two cmdlets:

- The `Export-Alias` cmdlet
- The `Import-Alias` cmdlet

In this exercise, we will create an alias for the `Test-Connection` cmdlet

and use it in other machines. The `Test` part will be the alias of `Test-Connection`. Following is the command to set the alias' name:

```
New-Alias -Name Test -Value Test-Connection -Description "Testing" -Verbose
```

Let's test the functionality using the following command:

```
test localhost
```

This command returns the following output:

Destination	IPV4Address	IPV6Address	Bytes	Time(ms)
localhost	127.0.0.1	::1	32	0
localhost	127.0.0.1	::1	32	0
localhost	127.0.0.1	::1	32	0
localhost	127.0.0.1	::1	32	0

Let's use the `Export-Alias` cmdlet and export only this alias using the following command:

```
Export-Alias -Name test C:\Temp\CustomAlias.txt -Verbose
```

We will get the following output:

```
# Alias File
# Exported by : ChenV
# Date/Time : Tuesday, August 25, 2015 1:57:28 PM
# Computer : CHENV
"Test","Test-Connection","Testing","None"
```

Move this text file to another computer and use the `Import-Alias` cmdlet to do this.

Look at the following image:

```

PS C:\> Import-Alias C:\Temp\test.txt -Verbose ①
VERBOSE: Performing the operation "Import Alias" on target "Name: test Value: Test-Connection".
PS C:\> Get-Alias -Name Test ②
 CommandType      Name          Version      Source
-----      ----          -----      -----
 Alias        test -> Test-Connection

PS C:\> test localhost ④
Source      Destination    IPV4Address   IPV6Address      Bytes      Time(ms)
-----      -----          -----          -----          -----      -----
 CHENWIN8    localhost     127.0.0.1      ::1           32          0
 CHENWIN8    localhost     127.0.0.1      ::1           32          0
 CHENWIN8    localhost     127.0.0.1      ::1           32          0
 CHENWIN8    localhost     127.0.0.1      ::1           32          0

```

The steps marked in the image we just discussed are explained as follows:

- 1: Here, I used the `Import-Alias` cmdlet and have given the path where I copied the text file
- 2: Here, I used the `Get-Alias` cmdlet to verify the existence of the alias
- 3: This indicates the result—**test -> Test-Connection**
- 4: This verifies the functionality of the alias

Note

There is no direct cmdlet to remove an alias, such as `Remove-Alias`. So, how do we do this?

Alias is one of the items in the PowerShell drive. Yes! We can use the `Remove-Item` cmdlet to remove the alias. Execute the following command:

```
Get-Item Alias:\test
```

The command that we just discussed returns the following output:

```

PS C:\> Get-Item Alias:\test
 CommandType      Name          Version      Source
-----      ----          -----      -----
 Alias        test -> Test-Connection

```

So, you can simply use `Remove-Item` to delete the alias. Execute the command as shown in the following image:

```
PS C:\> Remove-Item Alias:\test -Verbose  
VERBOSE: Performing the operation "Remove Item" on target "Item: test".
```

```
PS C:\> |
```

Alias

Understanding expressions

Windows PowerShell supports regular expressions. We know that PowerShell is built using the .NET framework. So, it strongly supports the regular expressions in .NET.

Developers prefer to use regular expressions to solve complex tasks such as formatting display names, manipulating files, and so on.

Regex can either be used by comparing operators or implementing the .NET class.

Note

The MSDN reference link for the regex class is

<https://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex%28v=vs.110%29.aspx>.

In this topic, we will cover the following:

- Operators and comparison operators
- Implementing regex using the .NET class
- Where do we use regular expressions?

Before we proceed with regular expressions, let's explore a few things about operators.

To know about operators, we can run the following code:

```
help about_Operators -ShowWindow
```

The output is illustrated in the following image:



To retrieve help for all the operators in Windows PowerShell, you can run each of the following lines of code and explore their usage:

```
help about_Arithmetic_Operators
```

```
help about_Assignment_Operators  
help about_Comparison_Operators  
help about_Logical_Operators  
help about_Type_Operators  
help about_Split  
help about_Join  
help about_Redirection
```

Windows PowerShell has operators such as arithmetic operators, assignment operators, comparison operators, logical operators, redirection operators, split and join operators, type operators, format operators, static member operators, and so on.

As we are exploring regular expressions, we will focus only on the comparison operators.

Comparison operators are used to compare values and to find the matching pattern. By default, comparison operators are not case sensitive.

We can also perform case-sensitive pattern matching using `c` before the operators.

Let's consider both these in the following example:

```
#Case In-Sensitive  
"PowerShell" -match "PowerShell"
```

This command returns the output as true.

```
#Case Sensitive  
"PowerShell" -cmatch "powershell"
```

This command returns the output as false.

Now, it's time for us to use the .NET regular expressions in Windows PowerShell with the comparison operators

Note

The MSDN link for a quick reference guide to regular expressions is

<https://msdn.microsoft.com/en-us/library/az24scfc%28v=vs.110%29.aspx>

Here is a command that uses a regular expression to check whether the first three characters are digits in a given string.

For example, the given string is 123-456-ABC. Then, the command would be as follows:

```
"123-245-ABC" -match '^\\d{3}'
```

This command returns the output as true.

```
"EFG-245-ABC" -match '^\\d{3}'
```

This command returns the output as false.

Now, let's take a look at how the command that we just discussed works, in the following list:

- ^: This is the beginning of the string.
- \d: This is to check the digits.
- {3}: This matches the previous elements *n* times. In our case, it's three times.

Use the following regular expression to remove the white space characters from any given string:

```
"Power Shell" -replace '\\s' , ''
```

This returns PowerShell.

Tip

In case of replacing a string with null with the help of the replace operator, we can execute the following command:

```
"Power Shell" -replace '\\s'
```

There is no need to explicitly mention replacing a string with a non-white space character.

The given input string is Power Shell. The expression removes the white space between Power and Shell in the string and outputs the Powershell word. The white space is removed as explained in the following steps:

- In the preceding code, we used the comparison operator, `-replace`
- The `\s` character is the white space character in regex
- This is replaced with a non-white space character

We can swap strings using regular expressions.

Consider the given string as FirstName LastName:

```
#Given Name: FirstName LastName
#Required Output: LastName, FirstName
"FirstName LastName" -replace "([a-z]+)\s([a-z]+)" , '$2, $1'
```

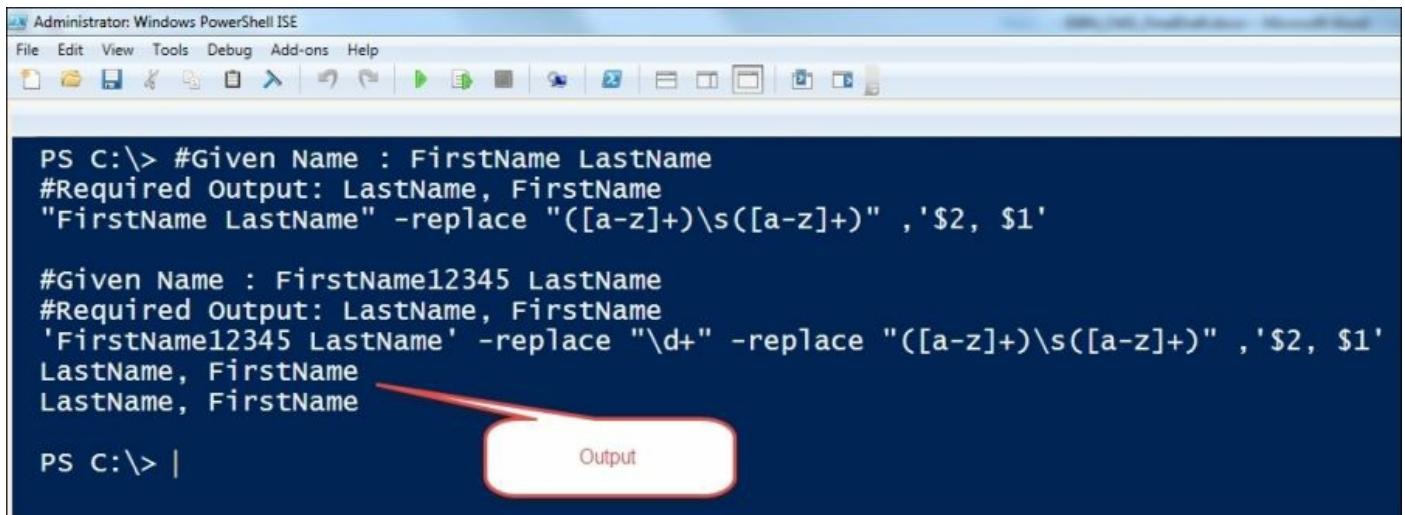
The commands that we just discussed return the output as LastName, FirstName.

Consider the given string as FirstName12345 LastName:

```
#Given Name: FirstName12345 LastName
#Required Output: LastName, FirstName
'FirstName12345 LastName' -replace "\d+" -replace "([a-z]+)\s([a-z]+)" , '$2, $1'
```

The commands that we just discussed return the output as LastName, FirstName.

The output of the previous two expressions is illustrated in the following image:



The screenshot shows a Windows PowerShell ISE window. The code pane contains the following PowerShell script:

```
PS C:\> #Given Name : FirstName LastName
#Required Output: LastName, FirstName
"FirstName LastName" -replace "([a-z]+)\s([a-z]+)" ,'$2, $1'

#Given Name : FirstName12345 LastName
#Required Output: LastName, FirstName
'FirstName12345 LastName' -replace "\d+" -replace "([a-z]+)\s([a-z]+)" ,'$2, $1'
LastName, FirstName
LastName, FirstName

PS C:\> |
```

A red arrow points from the word "Output" in a button at the bottom right of the code pane to the output pane below it.

Similarly, PowerShell allows us to use the `regex` class to perform the same tasks. Execute the following code:

```
[Regex]::IsMatch('PowerShell' , 'PowerShell')
```

Note

The MSDN TechNet article for the `regex` class is at the following link:

<https://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex%28VS.80%29.aspx>

The `IsMatch` method is a member of the `regex` class. This method is overloaded, which indicates whether the regular expression finds a match in the input string. This is a simple example to check whether a string contains another string.

Note

The MSDN TechNet article for the `regex` class members is at the following link:

https://msdn.microsoft.com/en-US/library/system.text.regularexpressions.regex_members%28v=vs.80%29.aspx

Developers can easily understand and implement `regex`. However, IT

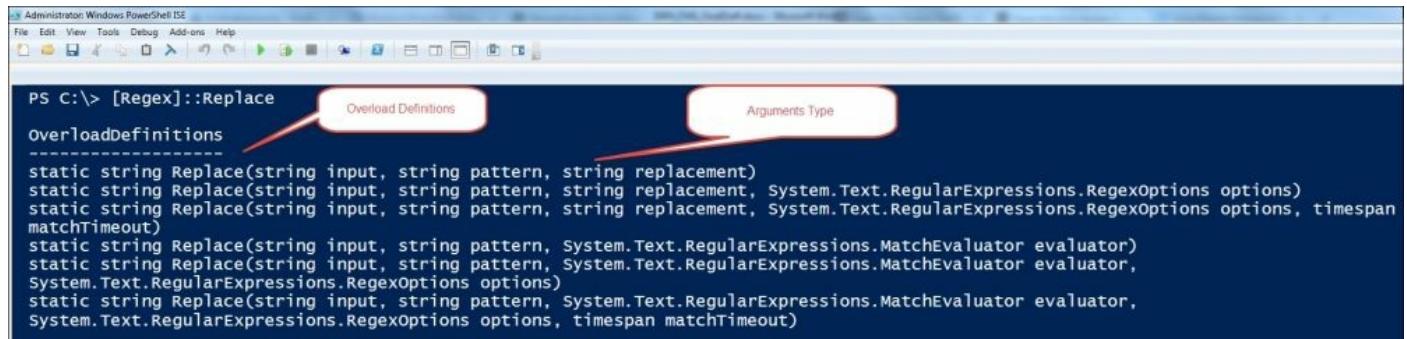
professionals or system administrators may have difficulties in understanding the arguments to be passed.

What arguments should we pass for members? It's not always necessary to refer to the MSDN article. Instead, we can use PowerShell to find the overloaded definitions.

In the following example, we will use the `Replace` method (the public method of the regex class):

```
[Regex] ::Replace
```

The output of the code that we just discussed is illustrated in the following image:



A screenshot of the Windows PowerShell ISE interface. The command `[Regex] ::Replace` is entered in the command line. A tooltip appears over the command, divided into two sections: "Overload Definitions" and "Arguments Type". The "Overload Definitions" section lists several static methods for the `Replace` function, each with different parameter signatures involving `string`, `RegexOptions`, and `MatchEvaluator`. The "Arguments Type" section is partially visible below it.

To remove the numbers from the given string using regex, we execute the following command:

```
[Regex] ::Replace('12String' , '\d{2}' , '')
```

The command that we just considered returns the output as `String`. We get this output as explained in the following steps:

- Using the `Regex` class, we invoked the public method, `Replace`.
- As per the overloaded definitions, we have an option to pass three arguments—string, pattern, and replacement string.
- The '`\d`' shorthand character represents the digits and `{n}` checks n times. In our case, it's two times.
- This was replaced with empty values.

Understanding objects

In general, a term object is something that we can touch and feel, and the same is applicable for PowerShell as well. An object in PowerShell is a combination of methods and properties:

$$\text{Objects} = \text{Properties} + \text{Methods}$$

A property is something that we can get or set. In short, properties store information about the object.

A method is an action to be invoked on a particular object.

Objects are constructed using their types, properties, and methods.

In this section, we will cover the following topics:

- Getting help about objects, properties, and methods
- Exploring objects, properties, methods, and types

Before we explore objects, let's have a look at the help documentation using the following commands:

```
help about_Objects  
help about_Properties  
help about_Methods
```

The objects that we see in PowerShell are a part of the .NET framework, and PowerShell will allow us to create custom objects as well.

Note

From Windows PowerShell 5.0 onward, we can create objects using a class. This will be covered in the [Chapter 2, Unleashing Development Skills Using Windows PowerShell 5.0](#).

Now, let's explore objects in detail. In the following example, we will use the `Get-Date` command:

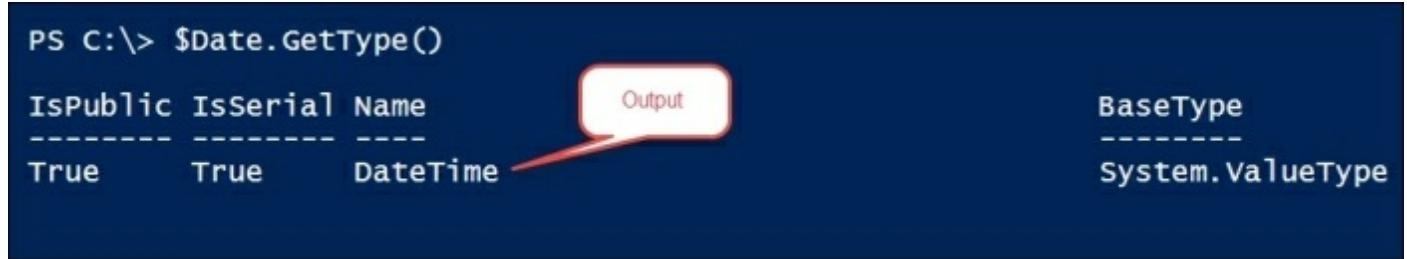
```
$Date = Get-Date
```

Here, \$Date is a variable in Windows PowerShell and Get-Date is a cmdlet to get the current date and time.

Once we run the preceding code, \$date will be a DateTime object. Let's take a look at the type of the \$Date variable:

```
$Date.GetType()
```

The output of this command is as follows:



IsPublic	IsSerial	Name	Output	BaseType
True	True	DateTime		System.ValueType

The Get-Member cmdlet is our friend, and helps us to explore the members and properties. To take a look at the available properties and methods, we can run the following code:

```
Get-Date | Get-Member -MemberType All -Force
```

The preceding command retrieves all the properties and methods.

The list will be huge; so, to view the properties and methods, we can change the MemberType value to either Property or Method. Let's execute the following code:

```
Get-Date | Get-Member -MemberType Property -Force
```

The output of this code is illustrated as follows:

```
PS C:\> Get-Date | Get-Member -MemberType Property -Force
```

TypeName: System.DateTime

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

The definitions of the property is shown as `{get;}`. Let's explore the property now, as follows:

```
(Get-Date).DateTime
```

The command that we just considered displays the current date and time of the local machine.

Now, let's take a look at how the command that we just discussed works, in the following list:

- `Get-Date`: This is the Windows PowerShell cmdlet.
- `..`: This is the property deference operator.
- The `DateTime` property shows the current system's date and time.

Alternatively, we can use the `DateTime` object, as follows:

```
[DateTime]::Now
```

We will get current date and time as output. For example, `Tuesday, June 02, 2015 12:15:51 PM`.

The operator used here is the static member operator, `::`.

To invoke the methods of an object, we will follow the same procedure. But, if the method needs arguments, we need to pass it accordingly, as shown in the following command:

```
Get-Date | Get-Member -MemberType Method -Force
```

Note

We used the `-Force` parameter to retrieve all the methods, including the hidden ones.

Execute the following code:

```
(Get-Date).AddDays(1)
```

Here, we added one day to the current day and the output is as follows:

```
Wednesday, June 03, 2015 12:55:35 PM
```

Alternatively, we can write `([DateTime]::Now).AddDays(1)`.

The `$psISE` object is the root object of the Integrated Scripting Environment. Using this we can toggle settings of the ISE, as follows:

```
$psISE | GM -Force
```

The output of the command we just discussed retrieves all the members of `$psISE`. One of the property options that hold all the options of the ISE is as follows:

```
$psISE.Options
```

To modify the zoom, use the following code:

```
$psISE.Options.Zoom = 150
```

To modify the Intellisense timeout seconds, we execute the following code:

```
$psISE.Options.IntellisenseTimeoutInSeconds = 5
```

To change the script pane's background color, we execute the following code:

```
$psISE.Options.ScriptPaneBackgroundColor = 'Green'
```

Understanding pipelines

A Windows PowerShell pipeline is used to join two or more statements with a pipeline operator. The Pipeline operator is '|'.

We have used pipelines in previous examples; let's know about pipeline use case scenario.

In this section, we will cover the following:

- Using a pipeline operator
- Where to use a pipeline operator

Windows PowerShell is designed to use pipeline. Here's an example of pipelines:

```
Command1 | Command2 | Command3
```

Here, `Command1` sends the object to `Command2`; the processed object will then be sent to `Command3`, which will output the results. Take a look at the following command:

```
help about_Pipelines -ShowWindow
```

A pipeline works in the following way:

- The parameter must accept input from a pipeline (however, not all do so)
- The parameter must accept the type of object being sent or a type that the object that can be converted to
- The parameter must not already be used in the command

Now, let's take a look at the following example:

```
PS C:\> Get-Service -Name BITS
Status      Name          DisplayName
```

Running BITS
Ser...

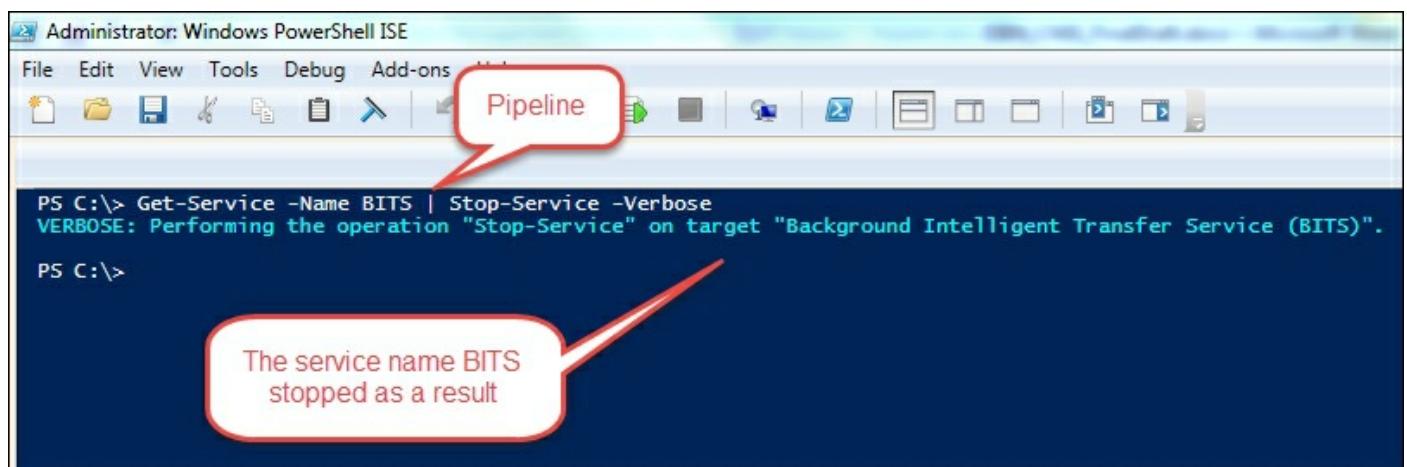
Background Intelligent Transfer

The `Get-Service` cmdlet gets the object representing the BITS service.

Using the `Stop-Service` cmdlet, we can stop the service. The `-Verbose` parameter is used to show the operation handled, as shown in the following code:

```
Get-Service -Name BITS | Stop-Service -Verbose
```

The output of the command we just discussed is illustrated in the following image:



```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Pipeline
PS C:\> Get-Service -Name BITS | Stop-Service -Verbose
VERBOSE: Performing the operation "Stop-Service" on target "Background Intelligent Transfer Service (BITS)".
PS C:\>
The service name BITS
stopped as a result
```

Using pipeline, we can do many more tasks, such as sorting, grouping, looping, and so on.

How do we find the parameter that accepts pipeline? Using help and pipeline, we can do this as shown in the following code:

```
help Get-Service -Parameter * | Select name , PipelineInput
```

The output of the command we just discussed is illustrated in the following image:

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

PS C:\> help Get-Service -Parameter * | Select name , PipelineInput

name	pipelineInput
ComputerName	true (ByPropertyName)
DependentServices	false
DisplayName	false
Exclude	false
Include	false
InputObject	true (ByValue)
Name	true (ByPropertyName, ByValue)
RequiredServices	false

Result!

PS C:\> |

To retrieve the Windows services of the remote machine, SharePoint001, we can write the command as follows:

```
'SharePoint001' | %{{Get-Service -ComputerName $} }
```

In short, the pipeline passes the output to another command so that the next command has something to work with or simply connects the output to other commands. This helps IT professionals automate tasks such as inventorying, reporting, and so on.

Exporting a running process to a CSV file

Let's take a look at the following command:

```
Get-Process | Export-csv C:\Temp\Process.csv  
-NoTypeInformation -Encoding UTF8
```

Refer to the following image:

The screenshot shows a Windows PowerShell ISE window with the title "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations like Open, Save, and Print. A tab bar at the top has "Untitled1.ps1*" and several other tabs numbered 2 through 5. The main code editor area contains the following PowerShell command:

```
1 1 Get-Process | Export-csv C:\Temp\Process.csv
2 -NoTypeInformation -Encoding UTF8
```

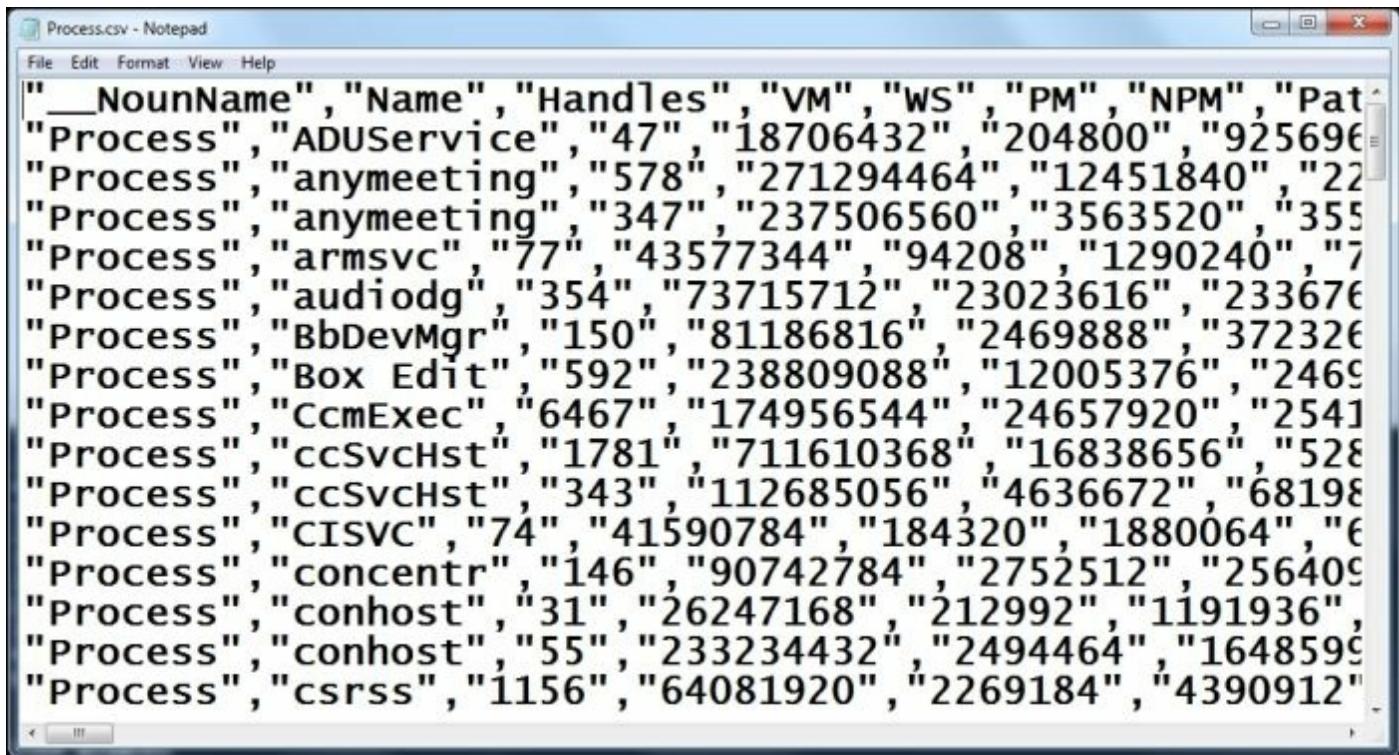
Annotations with red circles and numbers are placed over the code:

- 1: Points to the first character of "Get-Process".
- 2: Points to the vertical pipe character ("|") between "Get-Process" and "Export-csv".
- 3: Points to the first character of "Export-csv".
- 4: Points to the path "C:\Temp\Process.csv".
- 5: Points to the line continuation character at the end of the line.
- 6: Points to the "-NoTypeInformation" parameter.
- 7: Points to the "-Encoding" parameter and its value "UTF8".

The points marked in the image are explained as follows:

- 1: The `Get-Process` cmdlet is used to retrieve a running process in your local machine. Alternatively, the `gps` alias can be used
- 2: This is the pipeline operator used to pass the output to the next command
- 3: The `Export-csv` cmdlet is used to save the output in the `csv` format
- 4: This is the complete path of the output file
- 5: This is the PowerShell line continuation character
- 6: The `NoTypeInformation` switch parameter is used to avoid `#TypeInformation` in the `csv` header
- 7: The `Export-csv` cmdlet has an encoding parameter, which can be `ASCII`, `UTF7`, `UTF8`, `BigEndianUnicode`, `OEM`, or so on

The CSV output is shown in following image:



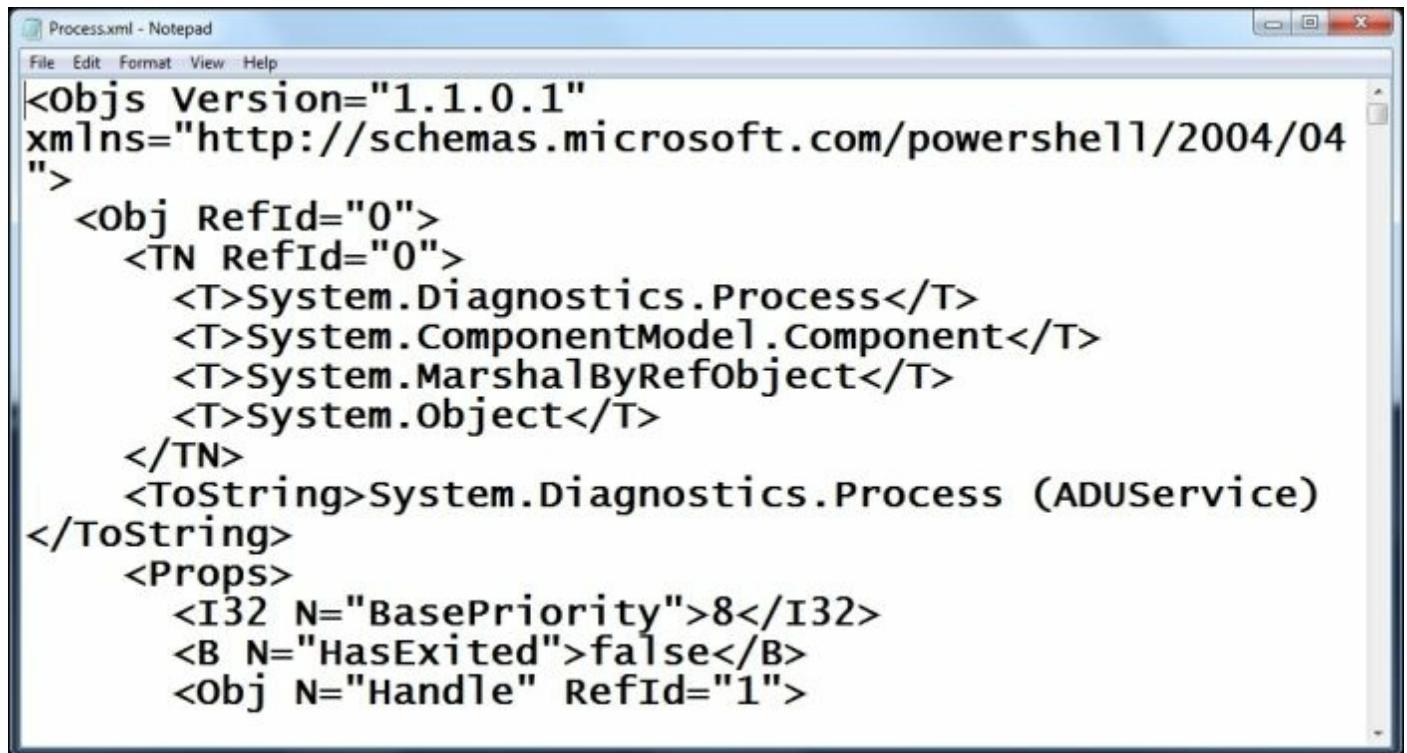
The screenshot shows a Windows Notepad window titled "Process.csv - Notepad". The menu bar includes File, Edit, Format, View, and Help. The main content area displays a CSV file with the following data:

__NounName	Name	Handles	VM	WS	PM	NPM	Pat
"Process",	"ADUService",	"47",	"18706432",	"204800",	"925696",		
"Process",	"anymeeting",	"578",	"271294464",	"12451840",	"22",		
"Process",	"anymeeting",	"347",	"237506560",	"3563520",	"355",		
"Process",	"armsvc",	"77",	"43577344",	"94208",	"1290240",	"7",	
"Process",	"audiodg",	"354",	"73715712",	"23023616",	"233676",		
"Process",	"BbDevMgr",	"150",	"81186816",	"2469888",	"372326",		
"Process",	"Box Edit",	"592",	"238809088",	"12005376",	"2469",		
"Process",	"CcmExec",	"6467",	"174956544",	"24657920",	"2541",		
"Process",	"ccSvCHst",	"1781",	"711610368",	"16838656",	"528",		
"Process",	"ccSvCHst",	"343",	"112685056",	"4636672",	"68198",		
"Process",	"CISVC",	"74",	"41590784",	"184320",	"1880064",	"6",	
"Process",	"concentr",	"146",	"90742784",	"2752512",	"256409",		
"Process",	"conhost",	"31",	"26247168",	"212992",	"1191936",		
"Process",	"conhost",	"55",	"233234432",	"2494464",	"1648599",		
"Process",	"csrss",	"1156",	"64081920",	"2269184",	"4390912",		

I prefer to export data in the XML format using the `Export-Clixml` cmdlet because it holds a lot more information. Take a look at the following command:

```
Get-Process | Export-Clixml C:\Temp\Process.xml
```

The output is shown in the following image:



The screenshot shows a Windows Notepad window titled "Process.xml - Notepad". The menu bar includes File, Edit, Format, View, and Help. The XML content is as follows:

```
<Objs Version="1.1.0.1"
xmlns="http://schemas.microsoft.com/powershell/2004/04">
<obj RefId="0">
<TN RefId="0">
<T>System.Diagnostics.Process</T>
<T>System.ComponentModel.Component</T>
<T>System.MarshalByRefObject</T>
<T>System.Object</T>
</TN>
<ToString>System.Diagnostics.Process (ADUService)
</ToString>
<Props>
<I32 N="BasePriority">8</I32>
<B N="HasExited">false</B>
<Obj N="Handle" RefId="1">
```

Using `Import-Clixml` and `Import-Csv`, we can view the output we exported:

```
Import-Clixml C:\Temp\Process.xml
```

The output of the command we just discussed is shown in the following image:

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

PS C:\> Import-Clixml C:\Temp\Process.xml

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
47	5	904	204	18	0.30	1800	ADUService
576	45	22084	12132	257	149.99	6092	anymeeting
347	33	34668	3484	227	3.37	6160	anymeeting
77	8	1260	96	42	0.03	1632	armsvc
353	12	22800	22480	70	0.75	15464	audiodg
150	16	3636	2416	77	2.14	6740	BbDevMgr
592	35	24060	11700	226	82.93	5812	Box Edit
6464	43	25028	24180	167	549.65	3260	CcmExec
1783	84	51784	16384	679	4,306.10	2632	ccSvcHst
345	28	6660	3820	107	94.58	4412	ccSvcHst
74	6	1836	180	40	0.09	2032	CISVC
146	13	2504	2696	87	0.61	6216	concentr
31	4	1164	212	25	0.08	2356	conhost
55	7	160996	2440	222	5.29	7320	conhost
1133	17	4288	2228	61	47.74	476	csrss
1109	30	4452	114232	317	934.95	588	csrss
410	25	17692	7168	98	195.87	1336	DcaSvc
414	29	24676	10052	206	3.56	148	DcaTray
160	22	70984	76804	317	...22.91	4052	dwm
1730	135	191384	108476	623	1,506.74	3740	explorer
142	14	5280	6556	92	...16.56	5620	FAHWindow
136	10	3140	940	38	0.53	1348	fdhost
53	6	1572	144	25	0.12	3296	fdlauncher
1328	187	764972	691556	1554	...37.20	11660	firefox

Using pipelines, we can connect multiple commands and get effective solutions, as explained in the following list:

- We can start, stop, or set a service
- We can export the output to report, for inventories, and so on.
- They connect commands and display the output as required
- They help in sorting, filtering, and formatting objects

Understanding filtering and formatting

In Windows PowerShell, filtering and formatting are used in most places to get the output in the desired format. `Select-Object` is very useful cmdlet to filter.

In this section, we will cover the following topics:

- Basics of filtering

- Basics of formatting

Consider the following command:

Get-Command -Noun Object

The output is as shown in the following image:

Use the following `Help` commands:

`Help Select-Object -Examples`
`Help Where-Object -Examples`

Using the `Select-Object` cmdlet, we can select the first and last n items from the collection of objects. The `Select-Object` cmdlet can be used to retrieve only unique values (ignoring duplicates).

Now, let's explore `Select-Object` for filtering. Let's consider that we have a set of objects from 65 to 90; to select the first 10, we need to pipe and use `Select-Object`, as shown in the following command:

65..90 | Select -First 10

We will get the output as 65 to 74.

Consider the following command:

1,2,2,3 | Select -Unique

Here, we will get the output as 1, 2, and 3.

Take a look at the following command:

```
1,2,2,3 | Select -Last 1
```

Here, we will get the output as 3.

Consider the following command:

```
1,2,2,3 | Select -SkipLast 1
```

We will get the output as 1, 2, and 2.

Take a look at the following command:

```
1,2,2,3 | Select -Skip 2
```

Here, we will get the output as 2 and 3.

To get help for all the parameters in `Select-Object`, use the following code:

```
help Select-Object -Parameter *
```

The `Where-Object` cmdlet is used to filter data returned by the other cmdlet. This cmdlet accepts comparison operators.

Let's explore the syntax of the `Where-Object` cmdlet, as follows:

```
(help Where-Object) .Syntax
```

The output of the command we just considered is illustrated in the following image:

The screenshot shows a Windows PowerShell ISE window with the title bar "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The main area displays the syntax for the `Where-Object` cmdlet, which is part of the Common Parameters section. The syntax is listed as a series of command-line examples, each starting with `Where-Object` followed by parameters like `-Property`, `<String>`, `[-Value] <Object>`, and various comparison operators (`-EQ`, `-NE`, `-GE`, etc.) and pipeline operators (`-InputObject`, `<PSObject>`, `-Contains`, etc.). The text is in a monospaced font.

```

PS C:\> (help Where-Object).Syntax

Where-Object [-Property] <String> [[-Value] <Object>] [-EQ] [-InputObject <PSObject>] [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -Contains [<CommonParameters>]
Where-Object [-FilterScript] <ScriptBlock> [-InputObject <PSObject>] [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -GE [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -GT [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -In [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CContains [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CEQ [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CGE [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CGT [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CIN [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CLE [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CLike [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CLT [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CMatch [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CNE [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CNotContains [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CNotIn [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CNotLike [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -CNotMatch [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -Is [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] - IsNot [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -LE [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -Like [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -LT [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -Match [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -NE [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -NotContains [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -NotIn [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -NotLike [<CommonParameters>]
Where-Object [-Property] <String> [[-Value] <Object>] [-InputObject <PSObject>] -NotMatch [<CommonParameters>]

```

Consider an array from 1 to 5. To select values greater than 3, we use the `Where-Object` alias, `?`, next to the pipeline operator, as shown in the following command:

```
1..5 | ? {$_ -gt 3}
```

From Windows PowerShell 4.0 onward, we can avoid pipelines for the `ForEach` and `Where` objects, as follows:

```
(65..90).ForEach({[char]$_})
```

The output of the code we just discussed is shown in the following image:

```
PS C:\> (65..90).ForEach({[char][int]$_})
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
```

Now, let's take a look at how this works in the following list:

- The `(65..90)` range uses the range operator, `..`; these are the ASCII values of A-Z
- We used the dereference operator, `.`, to invoke the `Foreach` method
- The `Foreach` method accepts expressions and arguments, as shown in the code we just considered.

Now, let's take a look at the following code:

```
(1..10).Where({$_. -ge 5})
```

The output of this code is shown in the following image:

The screenshot shows a Windows PowerShell ISE window. The command entered is `(1..10).Where({$_ -ge 5})`. The output is a list of numbers from 5 to 10. Three callout boxes highlight parts of the code:

- A red box labeled "Using Where Object" points to the `.Where()` method.
- A red box labeled "Condition" points to the expression `{$_ -ge 5}`.
- A red box labeled "Result >= 5" points to the output of the command, which is the result of the condition being true.

```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
PS C:\> (1..10).Where({$_ -ge 5})
5
6
7
8
9
10
PS C:\>
```

Now, let's take a look at how this works in the following list:

- The `(1..10)` range uses the range operator, `..`; this is the 1 to 10 array of the object
- We used the dereference operator, `.`, to invoke the `Where` method
- The `Where` method accepts the expression, mode, and number to return

We can use the `Where` statement with different modes. In the following example, we will select the first three values, where the number is greater than or equal to 5. Now, let's consider the following code:

```
(1..10).Where({$_ -ge 5}, 'First' , 3)
```

The output of this code is illustrated in the following image:

The screenshot shows a Windows PowerShell ISE window. The command entered is `(1..10).Where({$_ -ge 5}, 'First' , 3)`. The output is a list of numbers from 5 to 7. Three callout boxes highlight parts of the code:

- A red box labeled ">=5" points to the expression `{$_ -ge 5}`.
- A red box labeled "Select First" points to the mode parameter `'First'`.
- A red box labeled "Number to return" points to the value `3`.

```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
PS C:\> (1..10).Where({$_ -ge 5}, 'First' , 3)
5
6
7
PS C:\>
```

Now, let's take a look at how this works in the following list:

- The `(1..10)` uses the range operator, `..`, and this is the 1 to 10 array of the objects
- We used the dereference operator to invoke the `Where` method
- In this expression, `{$_ -ge 5}` is an object greater than 5, `First` is the mode, and `3` is the value for the number to be returned.

Similarly, we can use the `Split` mode as well. Using this, we can split the given collection of objects, as shown in the following code:

```
$section1 , $section2 = (1..100).Where({$_ -le 50} , 'Split' , 0)  
$section1  
$section2
```

The `$section1` variable contains values from 1 to 50, and the `$section2` variable contains values from 51 to 100.

Note

Avoid pipelines as much as you can. Use appropriately, because in larger script we may end up having a performance issue.

Use `Measure-Command` and analyze the performance of commands using pipelines.

Here is a table comparing commands using pipeline with those that don't use a pipeline:

With pipeline	Result in milliseconds	Without pipeline	Result in milliseconds
<code>65..90 %{ [char] [int] \$_ }</code>	4	<code>(65..90).ForEach({ [char] [int] \$_ })</code>	1
<code>1..10 ? {\$_ -ge 5} Select -First 3</code>	42	<code>(1..10).Where({\$_ -ge 5} , 'First' , 3)</code>	2

PowerShell formatting

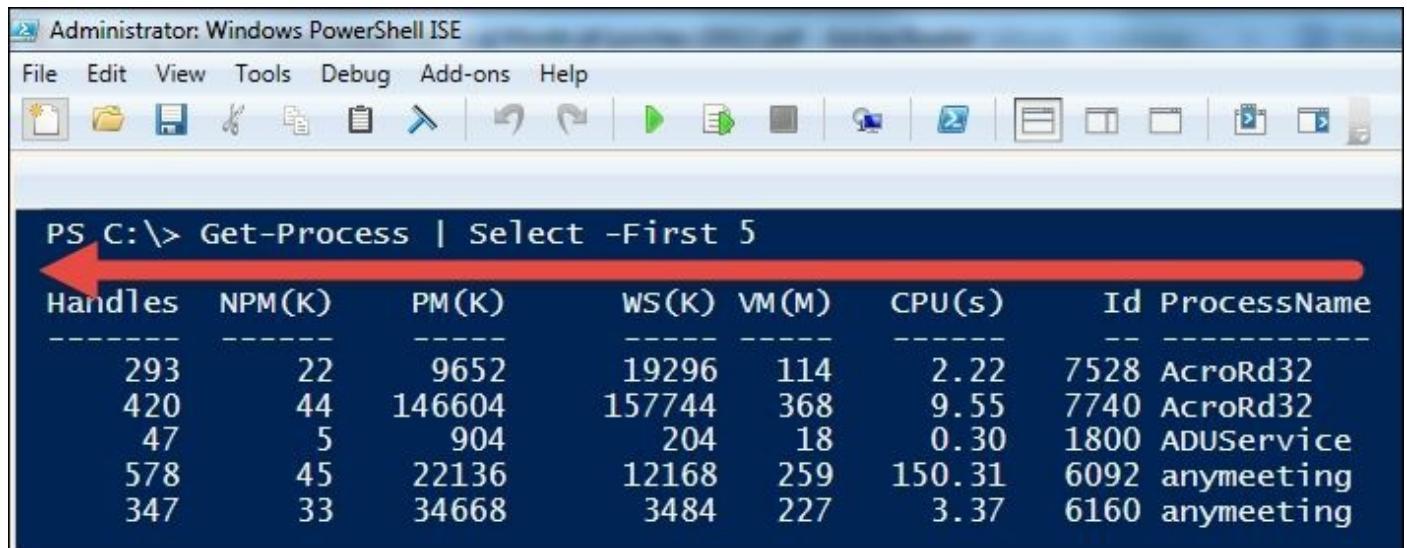
Windows PowerShell has a set of cmdlets that allows us to format the output. To find the cmdlets to format, use `Verb Format` to search, as shown in the following code:

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

Let's take a look at the default formatting of Windows PowerShell by executing the following the cmdlet:

```
Get-Process | Select -First 5
```

The output of this code is as follows:



A screenshot of the Windows PowerShell ISE interface. The title bar says "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The main window shows a command prompt with "PS C:\> Get-Process | Select -First 5". An arrow points from the left towards the command. The output is a table with the following data:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
293	22	9652	19296	114	2.22	7528	AcroRd32
420	44	146604	157744	368	9.55	7740	AcroRd32
47	5	904	204	18	0.30	1800	ADUService
578	45	22136	12168	259	150.31	6092	anymeeting
347	33	34668	3484	227	3.37	6160	anymeeting

The headers are not exactly property names. This formatting is done using the file name, `DOTNETTYPES FORMAT .PS1XML`.

The location of the file is `$PSHome`. Take a look at the following image:

```

        <PropertyName>PTEname</PropertyName>
        </TableColumnItem>
    </TableColumnItems>
</TableRowEntry>
</TableRowEntries>
</TableControl>
</View>
<View>
    <Name>process</Name>
    <ViewSelectedBy>
        <TypeName>System.Diagnostics.Process</TypeName>①
    </ViewSelectedBy>
    <TableControl>
        <TableHeaders>
            <TableColumnHeader>
                <Label>Handles</Label>
                <width>7</width>
                <Alignment>right</Alignment>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>NPM(K)</Label>②
                <width>7</width>
                <Alignment>right</Alignment>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>PM(K)</Label>③
                <width>8</width>
                <Alignment>right</Alignment>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>WS(K)</Label>
            </TableColumnHeader>
        </TableHeaders>
    </TableControl>
</View>

```

The following list explains the points marked in the preceding image:

- **1:** The type name of Get-Process is System.Diagnostics.Process
- **2:** For NPM(K), refer to the image preceding the previous image
- **3:** For PM(K), refer to the image preceding the previous image

The first thing we see in the XML file is the following warning:

Do not edit or change the contents of this file directly.
Please see the Windows PowerShell documentation or type.

Use the following command to obtain more information:

Get-Help Update-TypeData

So, let's not make any kind of modifications. Instead, let's take a look at the cmdlets to make minor modifications as desired, as shown in the following image:

CommandType	Name	Version	Source
Function	Format-Hex	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Format-Custom	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Format-List	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Format-Table	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Format-Wide	3.1.0.0	Microsoft.PowerShell.Utility

In the following example, we will use the `Where` method to select n items required to keep the output precise and short.

The `help` command to format the table is as follows:

```
help Format-Table -ShowWindow
```

Let's select the first five running services and format them as follows:

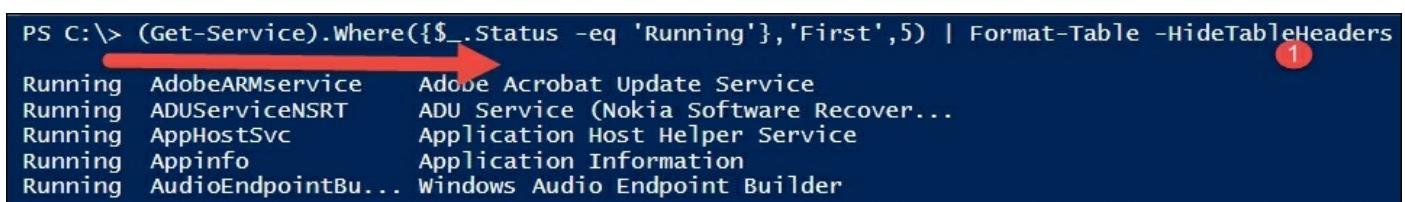
```
(Get-Service) .Where({$_.Status -eq 'Running'},'First',5) | Format-Table
```

This code outputs the default formatting.

Using the `Format-Table` cmdlet, we can hide the headers, auto size the table, use the expression, and much more. Let's consider the following command:

```
(Get-Service) .Where({$_.Status -eq 'Running'},'First',5) | Format-Table -HideTableHeaders
```

The output of the command we just discussed is shown in the following image:



PS C:\> (Get-Service) .Where({\$_.Status -eq 'Running'},'First',5) Format-Table -HideTableHeaders	1	
Running	AdobeARMservice	Adobe Acrobat Update Service
Running	ADUServiceNSRT	ADU Service (Nokia Software Recover...)
Running	AppHostSvc	Application Host Helper Service
Running	Appinfo	Application Information
Running	AudioEndpointBu...	Windows Audio Endpoint Builder

The default formatting of Windows PowerShell is not great, but it allows

us to customize the format as required. The report we deliver to IT Management should be precise and readable. Using Windows PowerShell, we can achieve this.

Reports can be in the HTML, XML, CSV, or other desired formats.

Using an expression is allowed in `Format-Table` inside the `{ }` script block token. Run the following command:

```
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" | Format-Table Name,@{n="Freespace(byte)" ;e="{0:N0}" -f $_.FreeSpace} ;a="center"
```

The following image shows all three alignments: left, right, and center:

The screenshot shows three separate PowerShell sessions demonstrating different alignment options for the `Freespace` column:

- Right Alignment:** The first session uses `a="right"`. The output shows the value `32,858,230,784` aligned to the right of the column.
- Left Alignment:** The second session uses `a="left"`. The output shows the value `32,858,161,152` aligned to the left of the column.
- Center Alignment:** The third session uses `a="center"`. The output shows the value `32,856,981,504` centered within the column.

Here is another example of a command to format the date in the day/month/year format using a format operator:

```
"Custom Date Format: {0},{1},{2}" -F (Get-Date).Day , (Get-Date).Month , (Get-Date).Year
```

To view the `LastWriteTime.DayOfWeek` file, run the following command:

```
Get-ChildItem C:\Temp | Ft name , @n="Day of Week" ; E =
```

```
{$__.LastWriteTime.DayOfWeek} }
```

The following is the command to custom format using the `Format-Custom` cmdlet:

```
Get-Service | Format-Custom
```

The default output of the command we just discussed is as follows:

```
class ServiceController
{
    Status = Running
    Name = wudfsvc
    DisplayName = Windows Driver Foundation - User-mode Driver Framework
}

class ServiceController
{
    Status = Stopped
    Name = WwanSvc
    DisplayName = WWAN AutoConfig
}

class ServiceController
{
    Status = Stopped
    Name = Zoho Assist-Remote Support
    DisplayName = Zoho Assist-Remote Support
}
```

Use the `Get-FormatData` cmdlet to view the formatting data from the `Format.ps1xml` formatting files. In this demo, we will try to customize the default format in the current session, as follows:

```
help Get-FormatData -ShowWindow
help Export-FormatData -ShowWindow
help Update-FormatData -ShowWindow
```

Perform the following steps:

Note

In this demo, we will change the column header to test. This will break the output ONLY in the current session.

1. Identify the type name of the command. For example, here, we will use the `Get-Process | GM` command.

2. The `TypeName` parameter is `System.Diagnostics.Process`, as shown in the following command:

```
Get-FormatData -TypeName System.Diagnostics.Process |  
Export-FormatData -Path C:\Temp\TestView.Format.ps1xml
```

3. Now, append the text of the column header in the `ps1xml` file in the `Temp` folder, as shown in the following command:

```
Update-FormatData -PrependPath  
C:\Temp\TestView.Format.ps1xml
```

4. The `Get-Process` command returns the output as shown in the following image:

The screenshot shows a PowerShell session with the following command and output:

```
PS C:\Windows\system32> Update-FormatData -PrependPath C:\temp\myprocessview.format.ps1xml -Verbose  
VERBOSE: Performing operation "Update FormatData" on Target "FileName: C:\temp\myprocessview.format.ps1xml"  
PS C:\Windows\system32> Get-Process
```

The output is a table showing process details. Two specific rows are highlighted with red boxes and arrows pointing to them:

Handles	NPM(K)	PM(Testing)	WS(K)	VM(Modified)	CPU(s)	ID	ProcessName
267	32487	Testing	1502	Modified	660	660	c2wtshost
265	1502		30		2732	2732	CcmExec
32	265		32		1980	1980	ccSvchst
35	30		35		7384	7384	ccSvchst
30	1578		30		1820	1820	conhost
1578	71		1578		2452	2452	conhost
71	328		71		10156	10156	conhost
328			328		10720	10720	conhost
					372	372	csrss
					432	432	csrss
					10596	10596	csrss

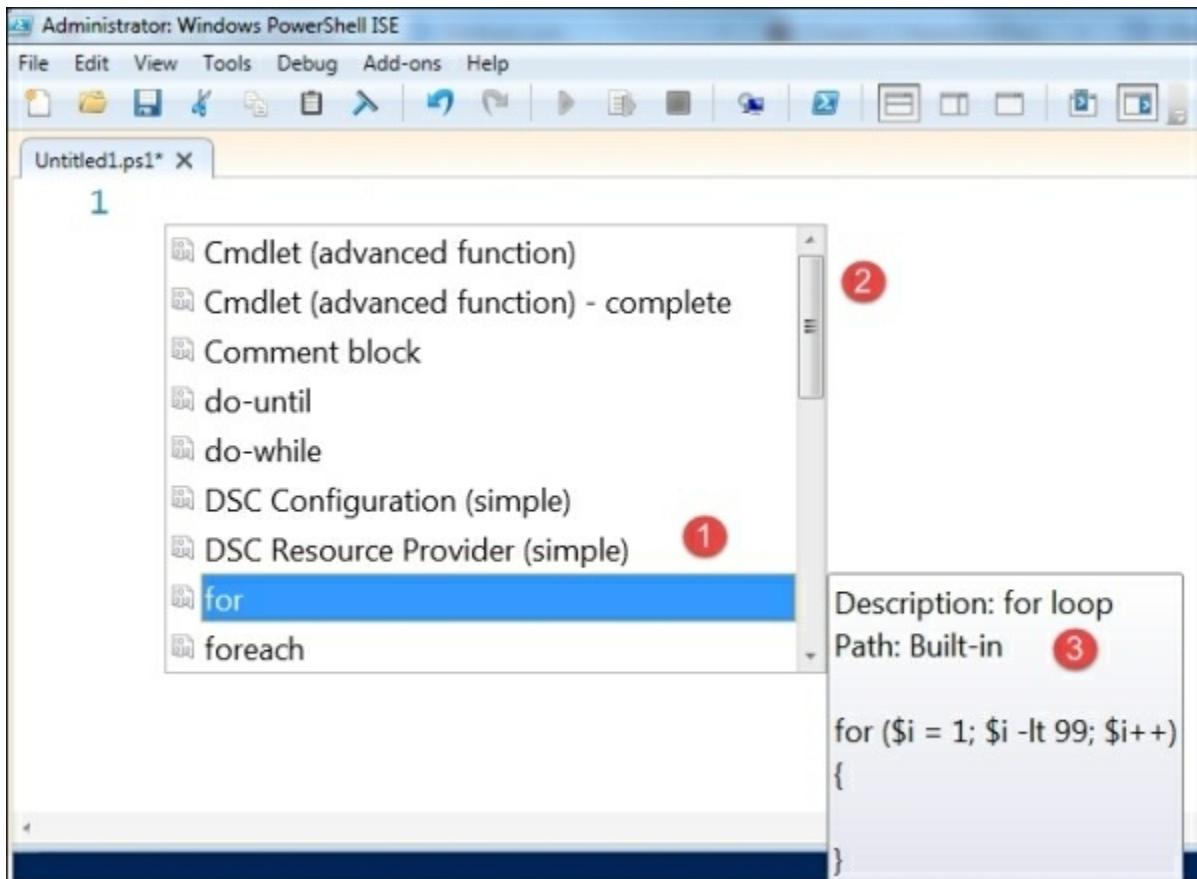
Exploring snippets in the PowerShell ISE

There is a cool way in the PowerShell ISE to reduce typing; however, before discussing PowerShell scripting, let's take a look at snippets.

Snippets are nothing but commonly used code. In PowerShell, we very often use functions, advanced functions, comment blocks, and so on.

Using an ISE, we don't type out the structure of the function. Instead, we can right-click on **Script Pane** and select **Start Snippets** or press **Ctrl + J**. This shows a menu of the available snippets.

The following image illustrates the snippets in the PowerShell ISE:



We need not be limited to only the available snippets; we can create our own snippets. In this demo, we will try to create a snippet. This is a simple snippet to add a mandatory parameter, which we can reuse in any of our functions.

This helps developers and IT professionals do the scripting faster.

There is no need for more typing; we just need to add a snippet wherever we need reusable codes, as shown in the following code:

```
$m = @'  
Param(  
[Parameter(Mandatory=$true)]  
[String]  
$String  
)  
'@  
New-IseSnippet -Text $m -Title Mandatory -Description 'Adds a  
Mandatory function parameter' -Author "Chen V" -Force
```

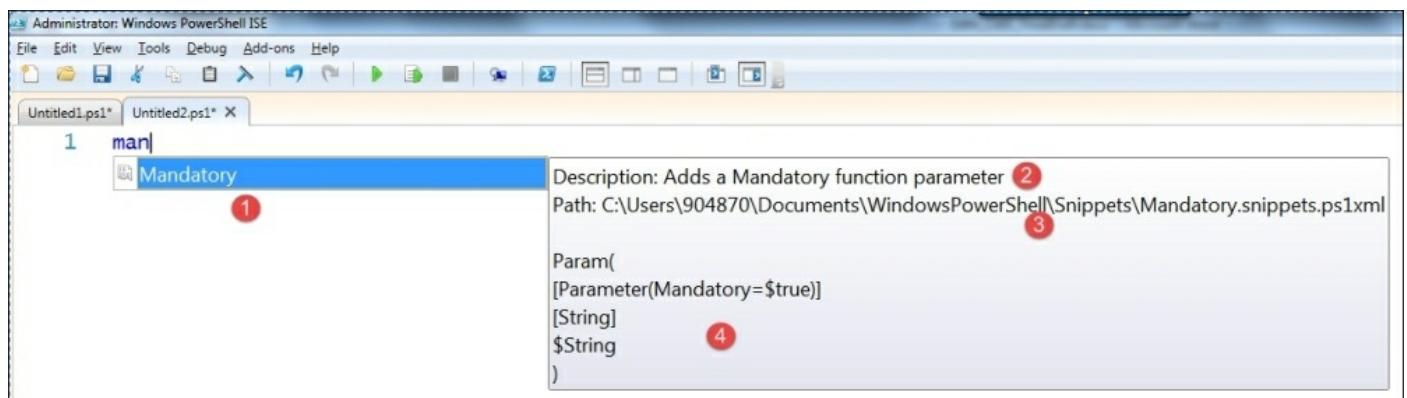
Let's take a look at how this works:

- The \$m variable holds the skeleton code
- The New-ISESnippet command is the one available in the module's ISE
- We used a few parameters with Text , which is the skeleton code; Title, which can be any friendly name; Description, which describes the snippet in short; and Author, which is the author's name
- After the code is executed, it creates a PS1XML file in the location, \$home\Documents\WindowsPowerShell\Snippets\
- The file name will be your title—in this case it's Mandatory.Snippets.PS1XML

Note

We can copy and place it any machine—pressing *Ctrl + J* will show Mandatory in the snippets.

The following image illustrates the output of the newly created snippet:



The following are the steps that explain the points marked in the preceding image:

- 1: The name Mandatory is the title of our new snippet
- 2: The **Description** field is shown in the pop-up box
- 3: The **Path** field is the path of the PS1XML file
- 4: This is the snippet code or skeleton code

Getting started with PowerShell scripting

Let's take a look at scripting in the cmdlet style.

We've arrived at a place from where we can explore PowerShell scripting with the knowledge of the previous topics. Wait! We haven't covered all that we need for scripting in PowerShell. Before we begin discussing scripting, we should know more about the scripting principles, using variables, commenting, writing help, and so on.

Here are a few principles of scripting:

If you want to deliver scripts to your organization or community, it's good to create the variables and follow the standard naming conventions. Do not use plain text passwords in the scripts, and avoid technical jargon in the comment blocks. Ensure that the script is readable for others.

Using Windows PowerShell scripting, we can perform complex tasks with the help of imperative commands. The scripting language supports branching, variables, and functions.

From now on, we will use the PowerShell ISE for its ease of use and benefits.

In this section, we will cover the following topics:

- Using variables
- The basics of Windows PowerShell scripting
- Writing the functions and advanced functions

Let's now discuss using variables.

A variable is used to store information, and it is the result of a running script.

Here's an example:

```
$value = Read-Host "Enter Value 1"
```

This script prompts for user input; once the value is entered, it stores it in the \$value variable, as shown in the following image:

The screenshot shows two windows. The top window is 'Administrator: Windows PowerShell ISE' with the title 'Code2.ps1*'. It contains the PowerShell command `$value = Read-Host "Enter Value 1"`. A red callout points from this line to a text box containing the annotation 'Read-Host cmdlet to prompt user inout'. The bottom window is a PowerShell session window titled 'PS C:\ScriptDemo>'. It shows the command `$value = Read-Host "Enter Value 1"` being run, followed by the user input '12'. A red callout points from the user input '12' to a text box containing the annotation 'User Input Value'. Another red callout points from the word 'Output' in the bottom-left corner of the session window to the user input '12'.

Note

Following are a few points to note about variables:

- They can be string or integer, and they allow special characters
- They should be used precisely—don't use special characters in their name

- You need to use the standard naming conventions that are easily understandable
- You can use the automatic, preference, and environment variables
- You should modify the preference variables only if required

Using the `New-Variable` cmdlet, a variable can be created along with a scope definition. In the following example, let's create a variable name, `ws`, which holds the value of the Windows service, and sets the scope to Global:

```
New-Variable -Name 'ws' -Value (Get-Service) -Scope Global
```

Using `Remove-Variable`, it can be removed, as shown in the following code:

```
Remove-Variable -Name ws -Verbose
```

Windows PowerShell scripting is used to automate your daily tasks. It may be anything such as reporting, server health checkup, performing tasks such as restarting services, stopping services, deploying solutions, installing Windows features, and much more.

The following points need to be considered while creating PowerShell scripting:

- Keep the PowerShell code simple and neat.
- Follow the same indentation throughout the code.
- Make a clear, comments-based help.
- Comment on your parameters with descriptions. This allows others to get help about the parameters.

Let's write a simple script that prints hello world on the screen:

```
#Windows PowerShell Script to Retrieve Windows Services
Write-Host "Hello, World!" -ForegroundColor Green
```

To run the PowerShell script, you need to call the script using a dot (.) operator followed by backward (\) slash.

The extension of the PowerShell script file should be `.PS1`.

The output of the preceding code is illustrated in the following image:

```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Code1.ps1 x
1 #Windows PowerShell Script to Retrieve Windows Services
2
3 Write-Host "Hello, World!" -ForegroundColor Green

PS C:\ScriptDemo> .\Code1.ps1
Hello, World!
```

Let's create a script that does a basic system inventory.

This PowerShell script will create a CSS file for styles, query the basic system information, convert to an HTML file, and open up after the script is completely executed, as shown in the following code:

```
$UserName = (Get-Item env:\username).Value
$ComputerName = (Get-Item env:\Computername).Value
$filepath = (Get-ChildItem env:\userprofile).value
Add-Content "$filepath\style.CSS" -Value " body {
font-family:Calibri;
font-size:10pt;
}
```

```

th {
background-color:black;
color:white;
}
td {
background-color:#19ffff0;
color:black;
}"
Write-Host "CSS File Created Successfully... Executing
Inventory Report!!! Please Wait !!!" -ForegroundColor Yellow
#ReportDate
$ReportDate = Get-Date | Select -Property DateTime |ConvertTo-
Html -Fragment
#General Information
$ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem |
Select -Property Model , Manufacturer , Description ,
PrimaryOwnerName , SystemType |ConvertTo-HTML -Fragment
#Boot Configuration
$BootConfiguration = Get-WmiObject -Class
Win32_BootConfiguration |
Select -Property Name , ConfigurationPath | ConvertTo-HTML -
Fragment
#BIOS Information
$BIOS = Get-WmiObject -Class Win32_BIOS | Select -Property
PSComputerName , Manufacturer , Version | ConvertTo-HTML -
Fragment
#Operating System Information
$SOS = Get-WmiObject -Class Win32_OperatingSystem | Select -
Property Caption , CSDVersion , OSArchitecture , OSILanguage |
ConvertTo-HTML -Fragment
#Time Zone Information
$TimeZone = Get-WmiObject -Class Win32_TimeZone | Select
Caption , StandardName |
ConvertTo-HTML -Fragment
#Logical Disk Information
$Disk = Get-WmiObject -Class Win32_LogicalDisk -Filter
DriveType=3 |
Select SystemName , DeviceID , @{Name="size(GB)";Expression={
{0:N1} -f($_.size/1gb)}}, @{Name="freespace(GB)";Expression={
{0:N1} -f($_.freespace/1gb)}} |
ConvertTo-HTML -Fragment
#CPU Information
$SystemProcessor = Get-WmiObject -Class Win32_Processor |
Select SystemName , Name , MaxClockSpeed , Manufacturer ,
status |ConvertTo-HTML -Fragment
#Memory Information

```

```

$PhysicalMemory = Get-WmiObject -Class Win32_PhysicalMemory |
Select -Property Tag , SerialNumber , PartNumber , Manufacturer ,
DeviceLocator , @{Name="Capacity(GB)";Expression={"{0:N1}" -f
($_.Capacity/1GB)}} | ConvertTo-Html -Fragment
#Software Inventory
$Software = Get-WmiObject -Class Win32_Product |
Select Name , Vendor , Version , Caption | ConvertTo-Html -
Fragment
ConvertTo-Html -Body "<font color = blue><H4><B>Report Executed
On</B></H4></font>$ReportDate
<font color = blue><H4><B>General Information</B></H4>
</font>$ComputerSystem
<font color = blue><H4><B>Boot Configuration</B></H4>
</font>$BootConfiguration
<font color = blue><H4><B>BIOS Information</B></H4></font>$BIOS
<font color = blue><H4><B>Operating System Information</B></H4>
</font>$OS
<font color = blue><H4><B>Time Zone Information</B></H4>
</font>$TimeZone
<font color = blue><H4><B>Disk Information</B></H4></font>$Disk
<font color = blue><H4><B>Processor Information</B></H4>
</font>$SystemProcessor
<font color = blue><H4><B>Memory Information</B></H4>
</font>$PhysicalMemory
<font color = blue><H4><B>Software Inventory</B></H4>
</font>$Software" -CssUri "$filepath\style.CSS" -Title "Server
Inventory" | Out-File "$FilePath\$ComputerName.html"
Write-Host "Script Execution Completed" -ForegroundColor Yellow
Invoke-Item -Path "$FilePath\$ComputerName.html"

```

The output of this code is illustrated in the following image:

General Information

Model	Manufacturer	Description	PrimaryOwnerName	SystemType
LIFEBOOK S761	FUJITSU	AT/AT COMPATIBLE	Windows User	x64-based PC

Boot Configuration

Name	ConfigurationPath
BootConfiguration	C:\windows

BIOS Information

PSComputerName	Manufacturer	Version
L06081	FUJITSU // Phoenix Technologies Ltd.	FUJ - 10e0000

Operating System Information

Caption	CSDVersion	OSArchitecture	OSLanguage
Microsoft Windows 7 Enterprise Service Pack 1	64-bit		1033

Time Zone Information

Caption	StandardName
(UTC+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna	W. Europe Standard Time

Disk Information

SystemName	DeviceID	size(GB)	freespace(GB)
L06081	C:	297.6	30.6

Let's take a look at how to write PowerShell functions with comments:

```
<#
.Synopsis
    To add two integer values
.DESCRIPTION
    Windows PowerShell Script Demo to add two values
    This accepts pipeline values
.EXAMPLE
    Add-Values -Param1 20 -Param2 30
.EXAMPLE
    12,23 | Add-Values
#>
function Add-Values
{
    [CmdletBinding()]
}
```

```

[Alias()]
[OutputType([int])]
Param
(
    # Param1 help description
    [Parameter(Mandatory=$true,
               ValueFromPipeline = $true,
               ValueFromPipelineByPropertyName=$true,
               Position=0)]
    #Accepts Only Integer
    [int]$Param1,
    #Accepts only integer
    [Parameter(Mandatory=$true,
               ValueFromPipeline = $true,
               ValueFromPipelineByPropertyName=$true,
               Position=0)]
    [int]$Param2
)
Begin
{
    "Script Begins"
}
Process
{
    $result = $Param1 + $Param2
}
End
{
    $result
}
}

```

This is not a good script, but I will use this to demonstrate a PowerShell function with comments in order to help explore the use of help.

The following is an explanation of how this code works:

1. The most important part is the comment block. Let's take a look at the following code snippet:

```

<#
.Synopsis
    To add two integer values
.DESCRIPTION
    Windows PowerShell Script Demo to add two values

```

```
This accepts pipeline values
.EXAMPLE
Add-Values -Param1 20 -Param2 30
.EXAMPLE
12,23 | Add-Values
#>
```

2. We need to provide a short description and synopsis of the script, as we did in this example. This will help others explore and know the usage of the script.
3. Then, we used the `Function` keyword and followed the standard *Verb-Noun* naming convention. In this example, we used `Add-Values`.
4. We parameterized our script using the `Param` block.
5. We named the parameter as applicable and added a comment/help description on top of the parameter.
6. We used the `Begin`, `Process`, and `End` blocks. I inserted the addition code in the `Process` block.

Note

To simplify all of these steps, open the ISE, press *Ctrl + J*, and select **CMDLET (advanced function)**.

To execute this script, we need to use the `.\
File name.PS1` command.
Take a look at the following image:

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

PS C:\ScriptDemo> . .\Code3.ps1 ①

PS C:\ScriptDemo> Get-Help Add-Values ②

NAME

 Add-Values ③

SYNOPSIS

 To add two integer values ④

SYNTAX

 Add-Values [-Param1] <Int32> [-Param2] <Int32> [<CommonParameters>] ⑤

DESCRIPTION

 Windows PowerShell Script Demo to add two values
 This accepts pipeline values ⑥

RELATED LINKS ⑦

REMARKS

 To see the examples, type: "get-help Add-Values -examples".
 For more information, type: "get-help Add-Values -detailed". ⑧
 For technical information, type: "get-help Add-Values -full".

PS C:\ScriptDemo>

The following is an explanation of the steps marked in the image we just considered:

- ①: We executed the Code3 script
- ②: We used the `Get-Help` cmdlet to read about the custom function
- ③: This is the name of the command
- ④: This is the customized synopsis
- ⑤: The `SYNTAX` field is autogenerated
- ⑥: This is the customized description
- ⑦: The `RELATED LINKS` field is empty because we haven't included it in our comment block
- ⑧: The `REMARKS` field is default

To view only the examples, we will use the following command:

Get-Help Add-Values -Examples

The output of this code is illustrated in the following image:

A screenshot of a PowerShell window titled "PS C:\ScriptDemo> Get-Help Add-Values -Examples". The help output is as follows:

```
NAME
  Add-Values ②

SYNOPSIS
  To add two integer values ③

----- EXAMPLE 1 -----
PS C:\>Add-Values -Param1 20 -Param2 30

----- EXAMPLE 2 -----
PS C:\>12,23 | Add-Values ④
```

The image highlights four points with red circles and numbers:

- ①: The `Get-Help` cmdlet is used to get help of the custom function or script.
- ②: This is the name of the cmdlet/function.
- ③: This shows EXAMPLE 1.
- ④: This shows EXAMPLE 2.

The following is an explanation of the points marked in the preceding image:

- ①: The `Get-Help` cmdlet is used to get help of the custom function or script.
- ②: This is the name of the cmdlet/function.
- ③: This shows EXAMPLE 1.
- ④: This shows EXAMPLE 2.

Similarly, we can view only parameters. The help description given above each parameter appears using the following code:

```
Get-Help Add-Values -Parameter *
```

Take a look at the following image:

```

PS C:\ScriptDemo> Get-Help Add-Values -Parameter *
-Param1 <Int32>
    Param1 help description ①
    Accepts Only Integer

    Required?          true
    Position?          1
    Default value      0
    Accept pipeline input? true (ByValue, ByPropertyName)
    Accept wildcard characters? false

-Param2 <Int32>
    Accepts only interger ②

    Required?          true
    Position?          1
    Default value      0
    Accept pipeline input? true (ByValue, ByPropertyName)
    Accept wildcard characters? false

```

The following are a few commands that use the `help` command:

```

help about_Functions
help about_Functions_Advanced_Methods
help about_Functions_Advanced_Parameters
help about_Functions_CmdletBindingAttribute
help about_Functions_OutputTypeAttribute

```

Let's take a look at how to write advanced functions.

Advanced functions are similar to compiled cmdlets but not exactly the same.

Here is a table comparing advanced functions and compiled cmdlets:

Advanced functions	Compiled cmdlets
These are designed using the PowerShell script.	These are designed using a .NET framework, such as C#, and compiled as DLL.
The actual work will be done in the process blocks.	The actual work will be done in the process

	records.
An advanced function can be created easily using the PowerShell ISE.	A compiled cmdlet can be created easily using the Visual Studio class library.
The performance of advanced functions is slower compared to a compiled cmdlet.	Compiled cmdlets are faster.

In this section, we will discuss both writing an advanced function code using the PowerShell ISE and a compiled cmdlet using the Visual Studio C# class library.

Let's take a look at how to create a compiled cmdlet using the Visual Studio C# class library. Execute the following code:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Management;
using System.Management.Automation;
using System.IO;
namespace Windows_Management
{
    [Cmdlet(VerbsCommon.Clear, "TemporaryInternetFiles")]
    public class WindowsManagement : PSCmdlet
    {
        protected override void ProcessRecord()
        {
            //Delete Internet Cache Files and Folders
            string path =
                Environment.GetFolderPath(Environment.SpecialFolder.InternetCache);
            Console.ForegroundColor = ConsoleColor.DarkYellow;
            Console.WriteLine("Clearing Temporary Internet
Cache Files and Directories....." + path);
            System.IO.DirectoryInfo folder = new
            DirectoryInfo(path);
            foreach (FileInfo files in folder.GetFiles())
            {
                try
                {
                    files.Delete();
                }
            }
        }
    }
}

```

```

        catch (Exception ex)
        {
            System.Diagnostics.Debug.WriteLine(ex) ;
        }
    }

    foreach ( DirectoryInfo Directory in
folder.GetDirectories())
    {
        try
        {
            Directory.Delete();
        }
        catch (Exception ex)
        {
            System.Diagnostics.Debug.WriteLine(ex) ;
        }
    }

    Console.WriteLine("Done Processing!!!!");
    Console.ResetColor();
}

}

namespace clearInternetexplorerHistory
{
    [Cmdlet(VerbsCommon.Clear, "IEHistory")]
    public class clearInternetexplorerHistory : PSCmdlet
    {
        protected override void ProcessRecord()
        {
            // base.ProcessRecord();
            string path =
Environment.GetFolderPath(Environment.SpecialFolder.History);
            Console.ForegroundColor = ConsoleColor.DarkYellow;
            Console.WriteLine("Clearing Internet Explorer
History....." + path);
            System.IO.DirectoryInfo folder = new
DirectoryInfo(path);
            foreach (FileInfo files in folder.GetFiles())
            {
                try
                {
                    files.Delete();
                }
                catch (Exception ex)
                {
                    System.Diagnostics.Debug.WriteLine(ex) ;
                }
            }
        }
    }
}

```

```

        }
    }

    foreach ( DirectoryInfo Directory in
folder.GetDirectories() )
    {
        try
        {
            Directory.Delete();
        }
        catch ( Exception ex)
        {
            System.Diagnostics.Debug.WriteLine(ex);
        }
    }

    Console.WriteLine("Done Processing!!!!");
    Console.ResetColor();
}

}

namespace UserTemporaryFiles
{
    [Cmdlet(VerbsCommon.Clear, "UserTemporaryFiles")]
    public class UserTemporaryFiles : PSCmdlet
    {
        protected override void ProcessRecord()
        {
            //base.ProcessRecord();
            string temppath = System.IO.Path.GetTempPath();
            System.IO.DirectoryInfo usertemp = new
DirectoryInfo(temppath);
            Console.WriteLine("Clearing Your Profile Temporary
Files..." + temppath);
            foreach ( FileInfo tempfiles in usertemp.GetFiles() )
            {
                try
                {
                    tempfiles.Delete();
                }
                catch ( Exception ex)
                {
                    System.Diagnostics.Debug.WriteLine(ex);
                }
            }

            Console.WriteLine("Done Processing!!!!");
            foreach ( DirectoryInfo tempdirectory in
usertemp.GetDirectories())

```

```

    {
        try
        {
            tempdirectory.Delete();
        }
        catch (Exception ex)
        {
            System.Diagnostics.Debug.WriteLine(ex);
        }
    }
}
}

```

Once the DLL is compiled, we can import it as a module in Windows PowerShell.

The output of the command we just discussed is illustrated in the following image:

PS C:\> Import-Module C:\ScriptDemo\WindowsMangement\bin\Debug\WindowsMangement.dll -Verbose
VERBOSE: Loading module from path 'C:\ScriptDemo\WindowsMangement\bin\Debug\WindowsMangement.dll'.
VERBOSE: Importing cmdlet 'Clear-TemporaryInternetFiles'.
VERBOSE: Importing cmdlet 'Clear-IEHistory'.
VERBOSE: Importing cmdlet 'Clear-UserTemporaryFiles'.
PS C:\> Get-Command Clear-TemporaryInternetFiles
CommandType Name Version Source
Cmdlet Clear-TemporaryInternetFiles 1.0.0.0 WindowsMangement

Get-Command

PS C:\> Get-Command Clear-TemporaryInternetFiles -Verbose
CommandType Name Version Source
Cmdlet Clear-TemporaryInternetFiles 1.0.0.0 WindowsMangement

Clear IE Temp

PS C:\> Clear-TemporaryInternetFiles -Verbose
Clearing Temporary Internet Cache Files and Directories.....C:\Users\904870\AppData\Local\Microsoft\Windows\Temporary Internet Files
Done Processing!!!

Now, let's consider the advanced functions in Windows PowerShell.

Advanced functions are more robust, can handle errors, support verbose and dynamic parameters, and so on.

Let's take a look at the small advanced functions used to retrieve system information, as follows:

```
function Get-SystemInformation
```

```

{
[CmdletBinding()]
[Alias()]
[OutputType([int])]
Param
(
    # Param1 help description
    [Parameter(Mandatory=$true,
               ValueFromPipelineByPropertyName=$true,
               ValueFromPipeline = $true,
               HelpMessage = "Enter Valid Host Names",
               Position=0)]
    [Alias('Host')]
    [ValidateCount(0,15)]
    [String[]]$ComputerName,
    # Param2 help description
    [int]
    $Param2
)
Begin
{
}
Process
{
    foreach($cn in $ComputerName)
    {
        $cs = Get-CimInstance -ClassName
Win32_ComputerSystem -ComputerName $cn
        $baseboard = Get-CimInstance -ClassName
Win32_BaseBoard -ComputerName $cn
        $properties = New-Object psobject -Property @{
            ComputerName = $cs.Caption
            Model = $cs.Model
            ComputerOwner = $cs.PrimaryOwnerName
            Bootupsate = $cs.BootupState
            BaseBoardSerialNumber = $baseboard.SerialNumber
            BaseBoardManufacturer = $baseboard.Manufacturer
        }
        $properties
    }
}
End
{
}
}

'localhost' , 'localhost' | %{$Get-SystemInformation -

```

```
ComputerName $_}
```

The output of the code we just discussed is illustrated in the following image:



```
BaseBoardManufacturer : Microsoft Corporation
Bootupstate          : Normal boot
BaseBoardSerialNumber : 0000-0012-0438-9235-0011-9062-41
Model                : Virtual Machine
ComputerOwner        : Employee
ComputerName         : WMF5NODE01

BaseBoardManufacturer : Microsoft Corporation
Bootupstate          : Normal boot
BaseBoardSerialNumber : 0000-0012-0438-9235-0011-9062-41
Model                : Virtual Machine
ComputerOwner        : Employee
ComputerName         : WMF5NODE01
```

As we have already covered the topic of adding help for PowerShell functions, I ignored it in the advanced functions section. Remember, we do have snippets in the ISE to create advanced functions. It's simple; just right-click on the script pane, select **Start snippet**, and then choose the **Advanced function complete** option.

Start documenting the synopsis, description, and help for parameters and build your code in the process block. This is very easy and handy to build advanced scripts using PowerShell.

Before building scripts, use the `Measure-Command` cmdlet and think about optimization. This will help in performance.

Summary

So far, we covered the very basics of Windows PowerShell. We now know the importance of an object-based shell and how an ISE helps us in building scripts. We explored the Windows PowerShell consoles and ISE snippets. After completing this chapter, you know the power of an interactive shell, how to use cmdlets effectively, how to use help, and the fundamentals of PowerShell along with building standard functions and creating cmdlets using the Visual Studio class library.

In the next chapter, we will cover **Common Information Model (CIM)**, **Windows Management Instrumentation (WMI)**, Extensible Markup Language, COM, .NET objects, modules, and a more exciting section, *Exploring Windows PowerShell 5.0*.

Chapter 2. Unleashing Development Skills Using Windows PowerShell 5.0

Windows PowerShell 5.0 is a dynamic and object-oriented scripting language. Compared to any other scripting language, it provides more benefits, such as reliability, security, managed code environments, and so on. Windows PowerShell has a strong connection with **Windows Management Instrumentation (WMI)**, **Common Information Model (CIM)**, **Extensible Markup Language (XML)**, and so on. Using this, we can develop solutions to automate our tasks.

PowerShell has the ability to manage different technologies. Using the PowerShell API, we can manage custom-built applications as well.

In this section, we will cover the following topics:

- Basics of WMI and CIM
- Exploring the XML and COM automation
- Exploring .NET objects for admins and development tasks
- Building advanced scripts and modules
- An insight into Windows PowerShell 5.0
- Script debugging

Basics of WMI and CIM

WMI—Windows Management Instrumentation—is the Microsoft implementation of WBEM—Web Based Enterprise Management—which allows us to access management information from any environment. PowerShell makes access to WMI easy and consistently deliverable using an object-based technique.

Let's explore a few PowerShell cmdlets of WMI.

WMI is a part of the `Microsoft.PowerShell.Management` module. You can run the following command to explore the WMI cmdlets:

```
Get-Command -Module Microsoft.PowerShell.Management -Name  
'*WMI*'
```

The output is illustrated in the following image:

PS C:\> Get-Command -Module Microsoft.PowerShell.Management -Name '*WMI*'			
CommandType	Name	Version	Source
Cmdlet	Get-WmiObject	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Invoke-WmiMethod	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Register-WmiEvent	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Remove-WmiObject	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Set-WmiInstance	3.1.0.0	Microsoft.PowerShell.Management

PS C:\>

There are many tools available to explore all the WMI classes available in the WMI repository. You can use the following links to do so:

<https://wmie.codeplex.com/>

<https://www.sapien.com/software/wmiexplorer>

Using these tools, we can explore and view the classes, instances, properties, and qualifiers easily. It's a GUI tool, so it makes our job easy as well. As we are focusing more on PowerShell, let's do it the PowerShell way, on the fly and explore.

The `Get-WmiObject` cmdlet has a switch parameter to list all the classes. The `Get-WmiObject -List` command will retrieve all the classes from the `Root\CIMV2` namespace by default. However, we can explicitly mention the namespace using the `NameSpace` parameter to identify the specified WMI class location, as shown in the following command:

```
#Retrieves WMI Class from Root\Security NameSpace  
Get-WmiObject -List -Namespace 'Root\Security'  
#Retrieves WMI Class from Root\CIMV2 - Default  
Get-WmiObject -List
```

To know more about WMI cmdlets, use the `help about_WMI_Cmdlets` command.

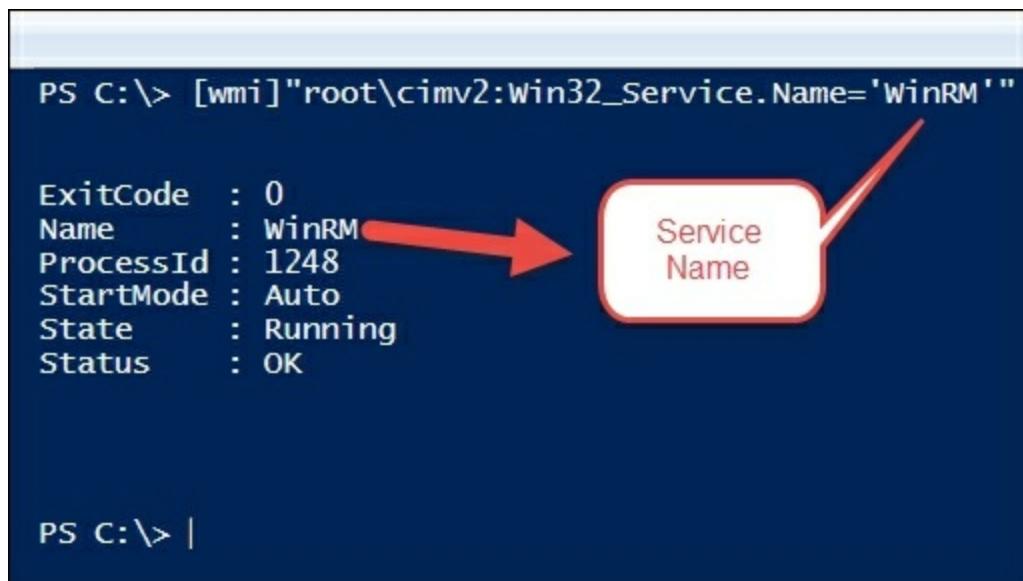
PowerShell supports the `WMIClass` type accelerators, which is a shortcut for using .NET classes in PowerShell.

- **[WMI]**: This is the type accelerator for the `ManagementObject` class
- **[WMIClass]**: This is the type accelerator for the `ManagementClass` class
- **[WMISearcher]**: This is the type accelerator for the `ManagementObjectSearcher` class

These make PowerShell richer and more useful; therefore, system administrators can use WMI in PowerShell very easily.

Let's query a service named `WinRM` using the WMI type accelerator. Run the following command:

```
[wmi]"root\cimv2:Win32_Service.Name='WinRM'"
```



```
PS C:\> [wmi]"root\cimv2:Win32_Service.Name='WinRM'"  
  
ExitCode : 0  
Name      : WinRM  
ProcessId : 1248  
StartMode  : Auto  
State     : Running  
Status    : OK  
  
PS C:\> |
```

The screenshot shows a PowerShell window with the command `[wmi]"root\cimv2:Win32_Service.Name='WinRM'"`. The output is a table of service properties. An annotation with a red arrow points from the `Name` column to a callout bubble containing the text "Service Name".

The same can be achieved using the following Windows PowerShell commands:

```
Get-WmiObject -Class Win32_Service -Filter "Name='WinRM'"  
Get-WmiObject -Class Win32_Service | ? {$_ .Name -eq 'WinRM'}  
(Get-WmiObject -Class Win32_Service).Where({$_ .Name -eq
```

```
'WinRM' })
```

All of the preceding commands provide the same result, and it depends upon the usage and optimization. The `Get-WMIObject` cmdlet has a parameter named `Query`, which allows us to use **WMI Query Language (WQL)**, as in the following command:

```
Get-WmiObject -Query "Select * from Win32_Service where Name='WinRM'"
```

A screenshot of a Windows PowerShell window. The command run is `Get-WmiObject -Query "Select * from Win32_Service where Name='WinRM'"`. The output shows the properties of the WinRM service, including ExitCode, Name, ProcessId, StartMode, State, and Status. A red arrow points from the command line to the output, and another red arrow points from the output back to the command line. A callout bubble labeled "Using Query Parameter" is positioned over the output area.

```
PS C:\> Get-WmiObject -Query "Select * from Win32_Service where Name='WinRM'"  
ExitCode : 0  
Name      : WinRM  
ProcessId : 1248  
StartMode  : Auto  
State     : Running  
Status    : OK
```

Using the `WMIClass` type accelerator, we can invoke any method easily, as shown in the following command:

```
$Obj = [wmiclass]"Win32_Process"  
$Obj.Create('NotePad.exe')
```

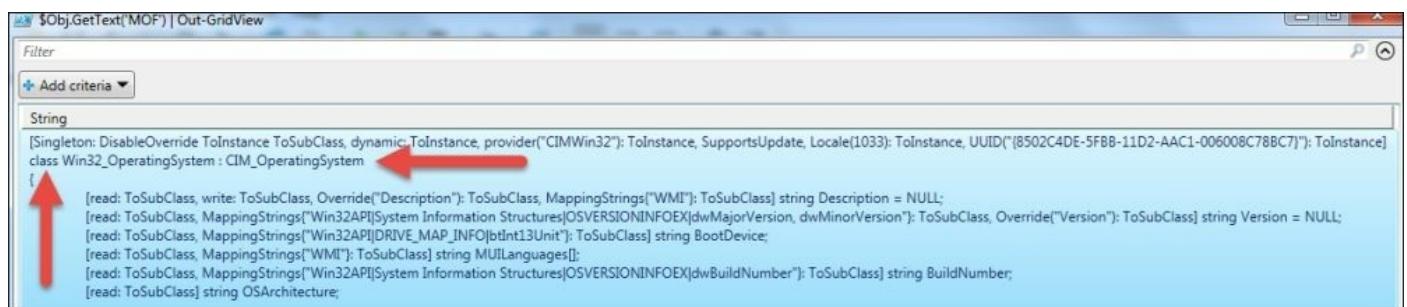
The output of this command is as shown in the following image:

A screenshot of the Windows PowerShell ISE. On the left, a script editor window titled "Untitled1.ps1" contains the following PowerShell code:
1 \$Obj = [wmiclass]"Win32_Process"
2
3 \$Obj.Create('NotePad.exe')
A red arrow points from the third line of the script to a callout bubble labeled "Create". On the right, a Notepad window titled "Untitled - Notepad" displays the text "Created Process Notepad.exe using WMIClass". A red arrow points from this text to a callout bubble labeled "Opened Notepad".

The WMI provider consists of the **Managed Object Format (MOF)** file, which defines the data, classes, and associated events. Using the WMIClass type accelerator method, it's possible to explore the MOF file. Consider the following commands:

```
$Obj = [wmiclass]"Win32_OperatingSystem"  
$Obj.GetText('MOF')
```

The preceding commands would output the MOF file as shown in the following image—you can use the `Out-GridView` cmdlet for look and feel:



```
Filter  
+ Add criteria ▾  
String  
[Singleton: DisableOverride ToInstance ToSubClass, dynamic ToInstance, provider("CIMWin32"); ToInstance, SupportsUpdate, Locale(1033); ToInstance, UUID("{8502C4DE-5FBB-11D2-AAC1-006008C78BC7}"); ToInstance]  
class Win32_OperatingSystem : CIM_OperatingSystem {  
    [read: ToSubClass, write: ToSubClass, Override("Description"): ToSubClass, MappingStrings["WMI": ToSubClass] string Description = NULL;  
    [read: ToSubClass, MappingStrings["Win32API\System\Information\Structures\OSVERSIONINFOEX\dwMajorVersion, dwMinorVersion": ToSubClass, Override("Version"): ToSubClass] string Version = NULL;  
    [read: ToSubClass, MappingStrings["Win32API\DRIVE_MAP_INFO\btInt13Unit": ToSubClass] string BootDevice;  
    [read: ToSubClass, MappingStrings["WMI": ToSubClass] string MUILanguages[];  
    [read: ToSubClass, MappingStrings["Win32API\System\Information\Structures\OSVERSIONINFOEX\dwBuildNumber": ToSubClass] string BuildNumber;  
    [read: ToSubClass] string OSArchitecture;
```

The same MOF file can be found at `$ENV:Windir\SYSTEM32\WBEM`. Use the following command:

```
Get-ChildItem C:\windows\System32\wbem -Filter *.MOF
```

Windows PowerShell has a command named `Invoke-WMIMethod`, which is used to invoke methods without using type accelerators.

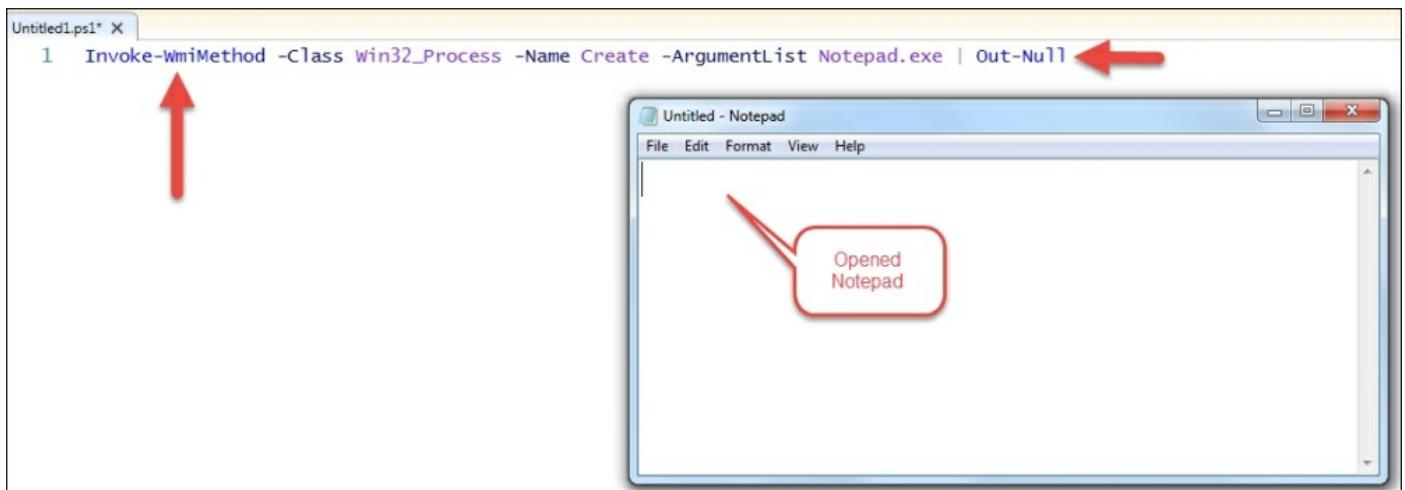
Note that in the following code, we have used `| Out-Null`, which deletes the output instead of sending it to the pipeline:

```
Invoke-WmiMethod -Class Win32_process -Name Create -  
ArgumentList Notepad.exe | Out-Null
```

This is similar to casting to `Void` type, as shown in the following command:

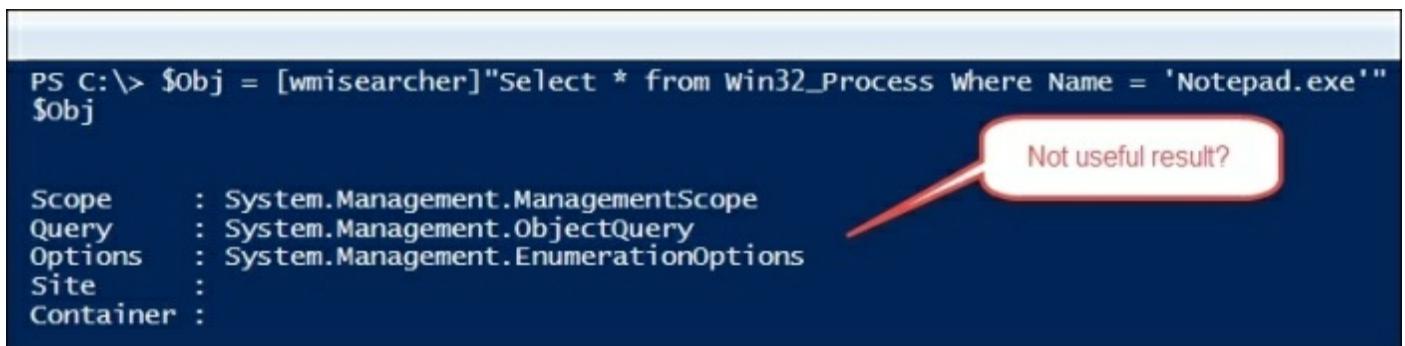
```
[Void] (Invoke-WmiMethod -Class Win32_process -Name Create -
```

ArgumentList Notepad.exe)



Using the `WMISearcher` type accelerator, we can explore the WMI data. Now, let's try the following code:

```
$Obj = [wmisearcher]"Select * from Win32_Process Where Name = 'Notepad.exe'"  
$Obj
```



Let's consider what this is. This outputs Scope, Query, Options, Site, and Container, which are the properties. To get the result collections we need to invoke `GetMethod()` method.

```
$Obj = [wmisearcher]"Select * from Win32_Process Where Name = 'Notepad.exe'"  
$Obj.Get()
```

Using pipelines, we can select the properties we need to view, as in the

following command:

```
$Obj = [wmisearcher]"Select * from Win32_Process Where Name = 'Notepad.exe'"  
$Obj.Get() | Select Caption , ExecutablePath , UserModeTime
```

The output is illustrated in the following image:

```
PS C:\> $Obj = [wmisearcher]"Select * from Win32_Process Where Name = 'Notepad.exe'"  
$Obj.Get() | Select Caption , ExecutablePath , UserModeTime ①  
Caption ExecutablePath ② UserModeTime  
----- -----  
notepad.exe C:\windows\SYSTEM32\notepad.exe 1560010  
notepad.exe C:\windows\system32\notepad.exe 6708043  
notepad.exe C:\windows\system32\Notepad.exe 312002  
notepad.exe C:\windows\system32\Notepad.exe 624004  
PS C:\> |
```

Output

The points marked in the figure are explained in the following list:

- 1: Here, we have used the pipeline to select the information required
- 2: This shows the output `Caption`, `ExecutablePath`, and `UserModeTime`

Let's consider a demo of a tiny PowerShell function using WMI.

We will need to retrieve the BIOS, computer system, and operating system information from the given servers in the environment.

We will use the `WIN32_BIOS`, `Win32_ComputerSystem`, and `Win32_OperatingSystem` classes to get the information.

Execute the following code:

```
Function Get-SystemInformation {  
    param(  
        [parameter(Mandatory = $true)]  
        [String]$computername
```

```

)
$OS = Get-WmiObject -Class Win32_OperatingSystem -
ComputerName $computername
$BIOS = Get-WmiObject -Class Win32_BIOS -ComputerName
$computername
$CS = Get-WmiObject -Class Win32_ComputerSystem -ComputerName
$computername
$properties = New-Object psobject -Property @{
    "OSName" = $os.Caption
    "ServicePack" = $os.CSDVersion
    "SerialNumber" = $BIOS.SerialNumber
    "Manufacturer" = $BIOS.Manufacturer
    "Bootupstate" = $cs.BootupState
}
$properties
}
Get-SystemInformation -computername localhost

```

```

① Function Get-SystemInformation {
    param( ③
        [parameter(Mandatory = $true)] ④
        [String]$computername ⑤
    )
    $OS = Get-WmiObject -Class Win32_OperatingSystem -ComputerName $computername
    ⑥ $BIOS = Get-WmiObject -Class Win32_BIOS -ComputerName $computername
    $CS = Get-WmiObject -Class Win32_ComputerSystem -ComputerName $computername
    $properties = New-Object psobject -Property @{}| ⑦
        "OSName" = $os.Caption
        "ServicePack" = $os.CSDVersion
        "SerialNumber" = $BIOS.SerialNumber ⑧
        "Manufacturer" = $BIOS.Manufacturer
        "Bootupstate" = $cs.BootupState
    }
    $properties ⑨
}

Get-SystemInformation -computername localhost ⑩

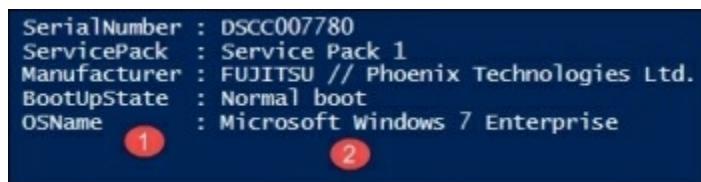
```

Let's consider how this works. Perform the following steps:

1. Use the `Function` keyword to create a function.
2. You can name the function `Get-SystemInformation` as this is a friendly name, but ensure that you follow the verb-noun combination for easy understanding.
3. Use the `Param` block to declare a variable.
4. Declare a variable and name it `$ComputerName` (For now, we will use a `localhost`).

5. Use `Win32_BIOS`, `Win32_ComputerSystem`, and `Win32_OperatingSystem` and assign each a variable.
6. Create an object using the `PSObject` class and collect all information in `$Properties` object.
7. Get the required properties.
8. Output the result.
9. Call the function.

The output is illustrated in the following image:



```

SerialNumber : DSCC007780
ServicePack : Service Pack 1
Manufacturer : FUJITSU // Phoenix Technologies Ltd.
BootUpState : Normal boot
OSName      : Microsoft Windows 7 Enterprise
  1          2

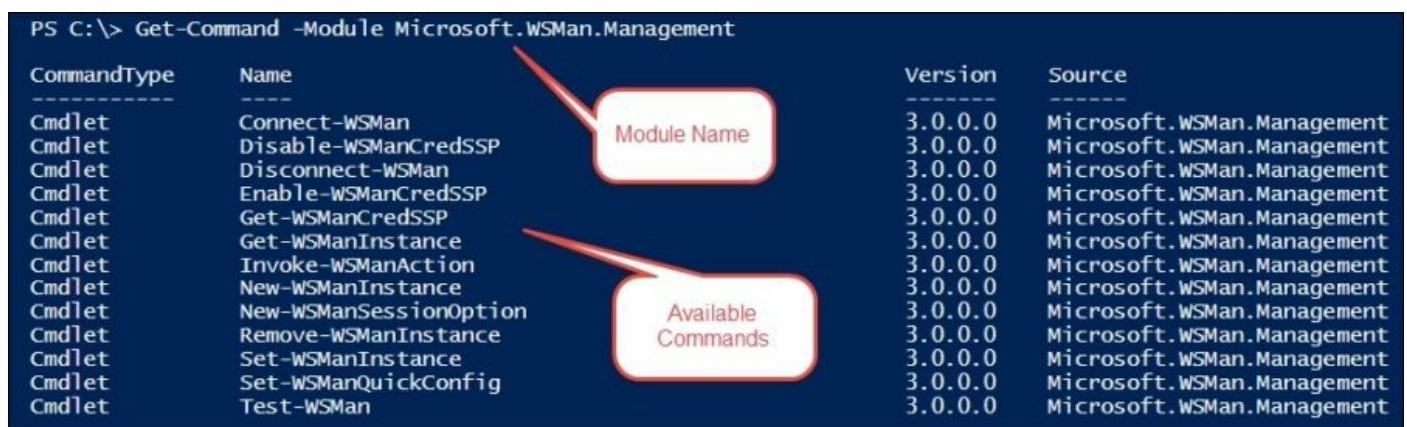
```

WMI uses **Distributed COM (DCOM)** to connect to a remote computer. However, in certain environments, this may be blocked by a firewall. In this scenario, we can retrieve the information using the PowerShell remoting feature or using the **WSMan** object.

To explore the WSMan commands, you can execute the following command:

```
Get-Command -Module Microsoft.WSMan.Management
```

The output is illustrated in the following image:



CommandType	Name	Version	Source
Cmdlet	Connect-WSMan	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Disable-WSManCredSSP	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Disconnect-WSMan	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Enable-WSManCredSSP	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Get-WSManCredSSP	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Get-WSManInstance	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Invoke-WSManAction	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	New-WSManInstance	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	New-WSManSessionOption	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Remove-WSManInstance	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Set-WSManInstance	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Set-WSManQuickConfig	3.0.0.0	Microsoft.WSMan.Management
Cmdlet	Test-WSMan	3.0.0.0	Microsoft.WSMan.Management

As we are discussing the basics of WMI here, we are not covering all the topics. However, we will discuss more about WMI and CIM in further topics.

CIM is defined by **Distributed Management Task Force (DMTF)** and is an object-oriented data model. In WMI, developers can use CIM to create classes. Using CIM, it's easy to manage the different elements of an environment.

CIM cmdlets are introduced in PowerShell 3.0, and these are vendor independent. Considering the recent Cloud operating system, we will have to work with different manufacturers. So, using CIM is the best option because WMI is Windows-based; it implements the DMTF standards in CIM and also allows us to query non-Windows operating systems.

The advantages of CIM in PowerShell are its usability and ability to run quite faster than WMI. PowerShell facilities such as tab completion make CIM cmdlets very rich. In other words, CIM is a superset of WMI.

To know the available CIM cmdlets, we can simply run the following code:

```
(Get-Command -Noun CIM*).Name
```

It will return the following output:

```
Get-CimAssociatedInstance
Get-CimClass
Get-CimInstance
Get-CimSession
Invoke-CimMethod
New-CimInstance
New-CimSession
New-CimSessionOption
Register-CimIndicationEvent
Remove-CimInstance
Remove-CimSession
Set-CimInstance
```

Let's query the basic OS and BIOS information as follows:

```
Function Get-SystemInformation {
    param(
        [Parameter(Mandatory = $true, ValueFromPipeline = $true)]
        [string[]]$ComputerName
    )
    Begin{}  

    Process{
        foreach($computer in $ComputerName) {
            $OS = Get-CimInstance -ClassName CIM_OperatingSystem -
ComputerName $computer
            $BIOS = Get-CimInstance -ClassName Win32_BIOS -
ComputerName $computer
            $props = New-Object psobject -Property @{
                OSName = $os.Caption
                ServicePack = $OS.CSDVersion
                BIOSSerialNumber = $BIOS.SerialNumber
                BIOSReleaseDate = $BIOS.ReleaseDate
            }
            $props
        }
    }
    End{}
}
#Demo
"localhost" , "localhost" | %{
    Get-SystemInformation -ComputerName $_
}
```

Note that every time you run the function, the output order is different. To get an ordered output, we can make a minor change in the code, as shown in the following:

```
$props = [Ordered] @{
    OSName = $os.Caption
    ServicePack = $OS.CSDVersion
    BIOSSerialNumber = $BIOS.SerialNumber
    BIOSReleaseDate = $BIOS.ReleaseDate
}
New-Object psobject -Property $props
```

The preceding code returns an ordered output, as shown in the following image:

OSName 1	ServicePack 2	BIOSSerialNumber	BIOSReleaseDate 4
Microsoft Windows 7 Enterprise	Service Pack 1	3	4/27/2011 2:00:00 AM
Microsoft Windows 7 Enterprise	Service Pack 1		4/27/2011 2:00:00 AM

The points marked in the figure are explained in the following list:

- **1:** OSName lists the names of operating systems
- **2:** ServicePack gives service pack information
- **3:** BIOSSerialNumber gives BIOS serial number information
- **4:** BIOSReleaseDate lists the release date of operating systems

What does [Ordered] do here? It simply makes an ordered dictionary. Let's take a look at another example of this.

The following code creates a hash table; the output order will be random:

```
$props = @{A='1'
    B='2'
    C='3'
}
```

To identify the type name, we use the GetType() method, which returns a hash table as names. Following is the command:

```
$props.GetType()
```

The following code will return an output in the same order (A, B, and C). This is an ordered dictionary type:

```
$props = [Ordered]@{A='1'
    B='2'
    C='3'
}
```

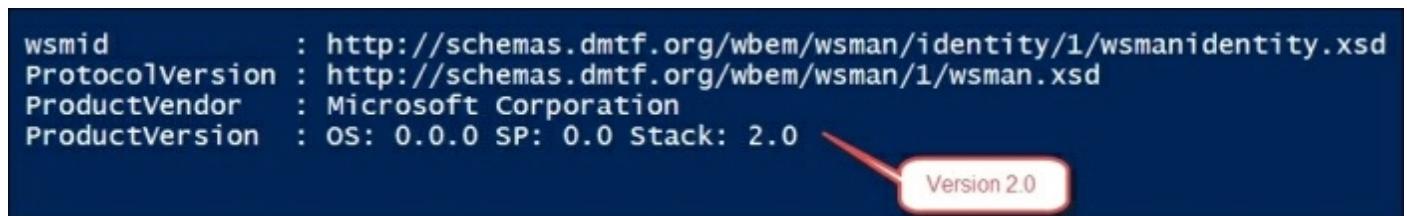
CIM cmdlets are introduced in PowerShell 3.0; so, before using CIM to query devices, we should ensure that it complies with the CIM and **WSMan** standards defined by DMTF. You may wonder, how can we use

CIM while querying in a mixed environment, where we may have Windows Server 2012 and 2008 R2 with PowerShell 2.0? The CIM class works on the devices that have PowerShell version 3.0. What happens if we try to use the `Invoke-Command` cmdlet? It will fail with an error message, '`Get-CimInstance`' is not recognized as the name of a cmdlet, function, script file, or operable program.

The solution is to use a CIM session. Let's take a look at how to use the CIM session in this example. Execute the following code:

```
Test-WSMan -ComputerName RemoteServer
```

The output is illustrated in the following image—the version used here is 2.0:



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

Version 2.0

All we need to know in the two CIM cmdlets are their parameters, which are as follows:

- The `New-CimSessionOption` cmdlet
- The `Get-CimInstance` cmdlet

Execute the following code:

```
$dcom = New-CimSessionOption -Protocol Dcom
$Remote = New-CimSession -ComputerName 'RemoteServer' -
SessionOption $dcom
(Get-CimInstance -CimSession $Remote -ClassName
CIM_OperatingSystem).InstallDate
```

Let's take a look at how this works.

We used the DCOM protocol, which is `New-CimSessionOption`, and assigned it to a `$dcom` variable. Then, we used the `New-CIMSession`

cmdlet to create a session with the `-SessionOption` parameter (we used `$dcom`, which is nothing but a `DComSessionOptions` type). Finally, we used the `Get-CimInstance` cmdlet and consumed `$Remote`, which is the `CimSession` type, with the DCOM protocol.

To remove the CIM session, we will simply use the following snippet:

```
Get-CimSession | Remove-CimSession
```

The benefits of using CIM cmdlets are as follows:

- There are loads of improvements while working with WMI association
- We can use `Get-CimClass` to explore a WMI class
- CIM session is beneficial to the query devices included with version 2.0
- There are no more DCOM errors

Working with XML and COM

We explored the basics of WMI and CIM; and yes, that was just the basics. After completing the XML topic, we will discuss **cmdlet definition XML (CDXML)**, which is used to map the PowerShell cmdlets, and the CIM class operations or methods.

I have seen most developers create XML files using Visual Studio and use some tools to compare XML. It's not a wrong method, but we have a much more convenient way to play with XML using PowerShell.

XML is the type accelerator for `System.Xml.Document`.

Note

To explore all the type accelerators in Windows PowerShell, use the following code:

```
[psobject].Assembly.GetType("System.Management.Automation.TypeAccelerators")::get
```

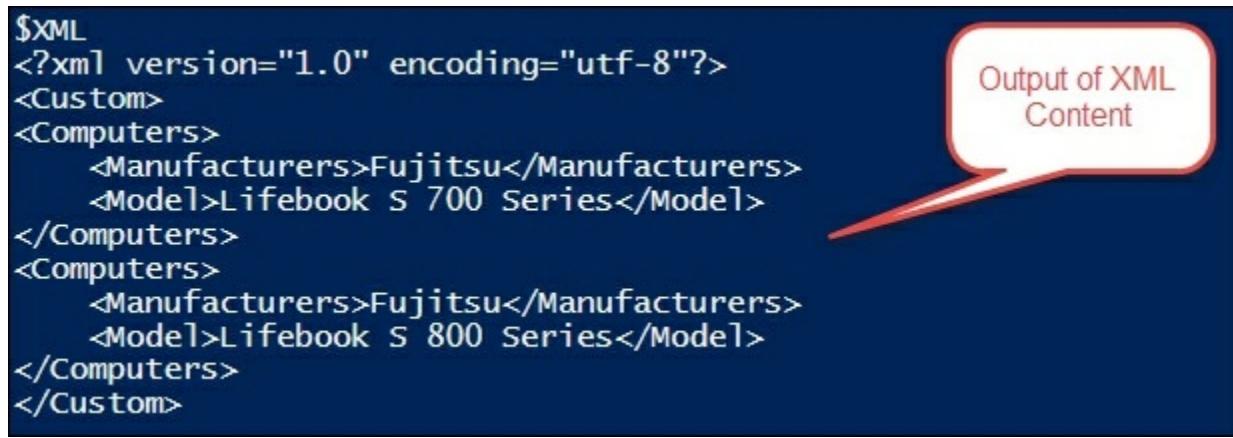
Let's take a look at the basic structure of an XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<Custom>
<Computers>
  <Manufacturers>Fujitsu</Manufacturers>
  <Model>Lifebook S 700 Series</Model>
</Computers>
<Computers>
  <Manufacturers>Fujitsu</Manufacturers>
  <Model>Lifebook S 800 Series</Model>
</Computers>
</Custom>
```

XML is case-sensitive, so it's always better to use some tools to create XML documents. I used Visual Studio 2013 to create this demo XML document.

Now, let's use the `Get-Content` cmdlet to call the XML document in

PowerShell, as shown in the following image:



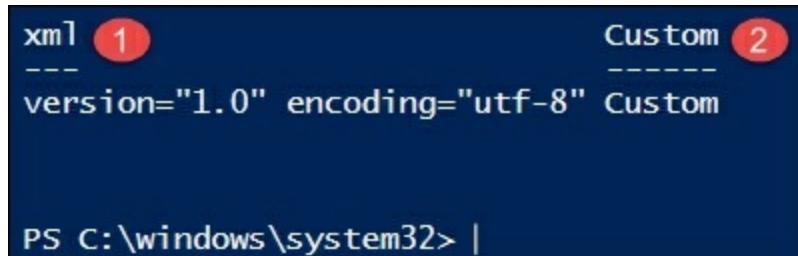
```
$XML
<?xml version="1.0" encoding="utf-8"?>
<Custom>
<Computers>
    <Manufacturers>Fujitsu</Manufacturers>
    <Model>Lifebook S 700 Series</Model>
</Computers>
<Computers>
    <Manufacturers>Fujitsu</Manufacturers>
    <Model>Lifebook S 800 Series</Model>
</Computers>
</Custom>
```

A red speech bubble points from the text "Output of XML Content" to the XML data.

Yeah! We can read this, but let's do in the XML way. So, let's use a type accelerator and perform our tasks, such as XML document manipulations, as in the following command:

```
[XML]$XML = Get-Content .\File1.XML
$xml
```

This outputs as shown in the following image:



```
xml 1
-----
version="1.0" encoding="utf-8" Custom 2
-----
```

The screenshot shows a PowerShell window. The command [XML]\$XML = Get-Content .\File1.XML is entered at the prompt. The output is displayed below it. Two red circles with numbers are overlaid on the output: circle 1 is over the word "xml" and circle 2 is over the word "Custom".

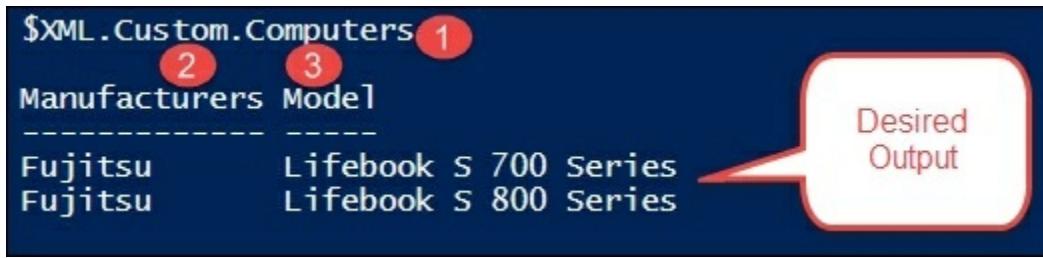
The points marked in the figure are explained in the following list:

- 1: This is the XML property
- 2: This is our element; it's the `XMLElement` property

We can quickly read the XML document with PowerShell using the following code:

```
[XML]$XML = Get-Content .\File1.xml
$xml.Custom.Computers
```

The output we need is illustrated in the following image:



The points marked in the figure are explained in the following list:

- The \$XML variable contains the XML document information, and now it is an object
- Manufacturers is a property that can be get and set
- Model is a property that can be get and set

Using PowerShell, we can read and manipulate the output as we need.

Now, let's take a look at the available properties, methods, and so on. Run the following command:

```
$XML.Custom.Computers | GM
```

This will output all the methods, properties, code methods, and parameterized properties, as shown in the following image:

PS C:\windows\system32> \$XML Get-Member -MemberType Methods		
Name	MemberType	Definition
ToString	CodeMethod	static string XmlNode(psobject instance)
AppendChild	Method	System.Xml.XmlNode AppendChild(System.Xml.XmlNode newChild)
Clone	Method	System.Xml.XmlNode Clone(), System.Object ICloneable.Clone()
CloneNode	Method	System.Xml.XmlNode CloneNode(bool deep)
CreateAttribute	Method	System.Xml.XmlAttribute CreateAttribute(string name), System.Xml.XmlAttribute
CreateCDataSection	Method	System.Xml.XmlCDataSection CreateCDataSection(string data)
CreateComment	Method	System.Xml.XmlComment CreateComment(string data)
CreateDocumentFragment	Method	System.Xml.XmlDocumentFragment CreateDocumentFragment()
CreateDocumentType	Method	System.Xml.XmlDocumentType CreateDocumentType(string name, string publicId, s
CreateElement	Method	System.Xml.XmlElement CreateElement(string name), System.Xml.XmlElement Creat
CreateEntityReference	Method	System.Xml.XmlEntityReference CreateEntityReference(string name)
CreateNavigator	Method	System.Xml.XPath.XPathNavigator CreateNavigator(), System.Xml.XPath.XPathNavi
CreateNode	Method	System.Xml.XmlNode CreateNode(System.Xml.XmlNodeType type, string prefix, str
CreateProcessingInstruction	Method	System.Xml.XmlProcessingInstruction CreateProcessingInstruction(string target
CreateSignificantWhitespace	Method	System.Xml.XmlSignificantWhitespace CreateSignificantWhitespace(string text)

Before we use methods, let's do an exercise using the XML file as the

configuration file that passes the parameters to the PowerShell scripts. The reason we need this is to simplify automation in environments. Based on our requirements, we can customize and code the script.

In this exercise, let's send an e-mail, where the parameters are preconfigured in an XML file, and the script reads the XML file to do the task.

The XML code is shown as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<Settings>
    <SMTP>SMTP NAME</SMTP>
    <MailFrom>From Address</MailFrom>
    <MailTO>To Address</MailTO>
    <Subject>XML Demo</Subject>
</Settings>
```

Using the following PowerShell code we can trigger emails:

```
[XML]$email = Get-Content .\File2.XML
$email = @{
    FROM = $email.Settings.MailFrom
    TO = $email.Settings.MailTO
    SMTP = $email.Settings.SMTP
    Subject = $email.Settings.Subject
}
Send-MailMessage @mail
```

PowerShell has a few commands to use XML; to find them, you can simply try the following code:

```
Get-Command *XML
```

The output is as shown in the following image:

PS C:\windows\system32> Get-Command *XML			
CommandType	Name	Version	Source
Cmdlet	ConvertTo-Xml	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Export-Clixml	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Import-Clixml	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Select-Xml	3.1.0.0	Microsoft.PowerShell.Utility

The points marked in the figure are explained in the following list:

- 1: Microsoft.PowerShell.Utility is the name of the module
- 2: Starting from PowerShell 5.0, the version appears in the output
- 3: Name shows all the available commands for XML
- 4: Command types, here shown as cmdlet

The `ConvertTo-Xml` cmdlet creates an XML-based representation of an object, and it's a .NET-based object. Try the following code in your system:

```
$hash = @{
    ServiceName = (Get-Service -Name Bits).Name
    Status = (Get-Service -Name Bits).Status
    CanStop = (Get-Service -Name Bits).CanStop
    CanPauseandContinue = (Get-Service -Name
    Bits).CanPauseAndContinue
}
$value = New-Object PSObject -Property $hash
($value | Convertto-xml -
NoTypeInformation).save("C:\Temp\Data.xml")
$xml = Get-Content C:\Temp\Data.xml
$xml.Objects.Object.Property
```

In the preceding code, we created a hash table using the `$hash` variable; then, we created a `PSObject` object and converted it to XML using the `ConvertTo-XML` command. The output is illustrated in the following image:

The screenshot shows the Windows PowerShell ISE interface with a script named `B04702_02_02.ps1`. The code in the editor is as follows:

```
1 $hash = @{
2     ServiceName = (Get-Service -Name Bits).Name
3     Status = (Get-Service -Name Bits).Status
4     CanStop = (Get-Service -Name Bits).CanStop
5     CanPauseandContinue = (Get-Service -Name Bits).CanPauseAndContinue
6 }
7
8 $value = New-Object PSObject -Property $hash
9 ($value | Convertto-xml -NoTypeInformation).save("C:\Temp\Data.xml")
10
11 $xml = Get-Content C:\Temp\Data.xml
12 $xml.Objects.Object.Property
```

Three callout boxes with arrows point to specific parts of the code:

- A box points to the first few lines: "Created a hash table which retrieves basic information of BITS service".
- A box points to the save command: "Converting to XML and invoking SAVE method".
- A box points to the final line: "Explore the Output".

This is similar to `ConvertTo-XML`; we can use the `Export-Clixml` command, and this does more or less the same job, except that this saves the results as an XML file. To import, we will use the `Import-Clixml` command. The `Export-Clixml` command is very helpful when we use credentials in the scripts, while exporting the credentials as XML and importing it to use in the scripts.

Note

Note that this is a secure way because `Export-Clixml` encrypts the password using the **Windows Data Protection API**. So, decryption works only with the user account that has encrypted the credentials.

Before we start our next exercise, we need to set up the `MSOnline` module in our machine. The installation involves just clicking on next and finish—we need to follow the installation wizard.

Perform the following steps:

1. First and foremost, download and install **Microsoft Online Services Sign-In Assistant** from <http://go.microsoft.com/fwlink/?linkid=236300>.
2. Then, download and install **Microsoft Online Services Module for Windows PowerShell** from <http://go.microsoft.com/fwlink/?linkid=236297>.

As we are done with the installation, let's connect to the **Exchange Online** session using PowerShell. Perform the following steps:

1. Export the credential using the `Export-Clixml` command, as follows:

```
$credentials = Get-Credential  
$credentials | Export-Clixml C:\Temp\SeccureString.xml
```

2. Import the credential using the `Import-Clixml` command, as follows:

```
$o365Cred = Import-Clixml C:\Temp\SeccureString.xml
```

3. Connect to the Exchange Online session using the following code:

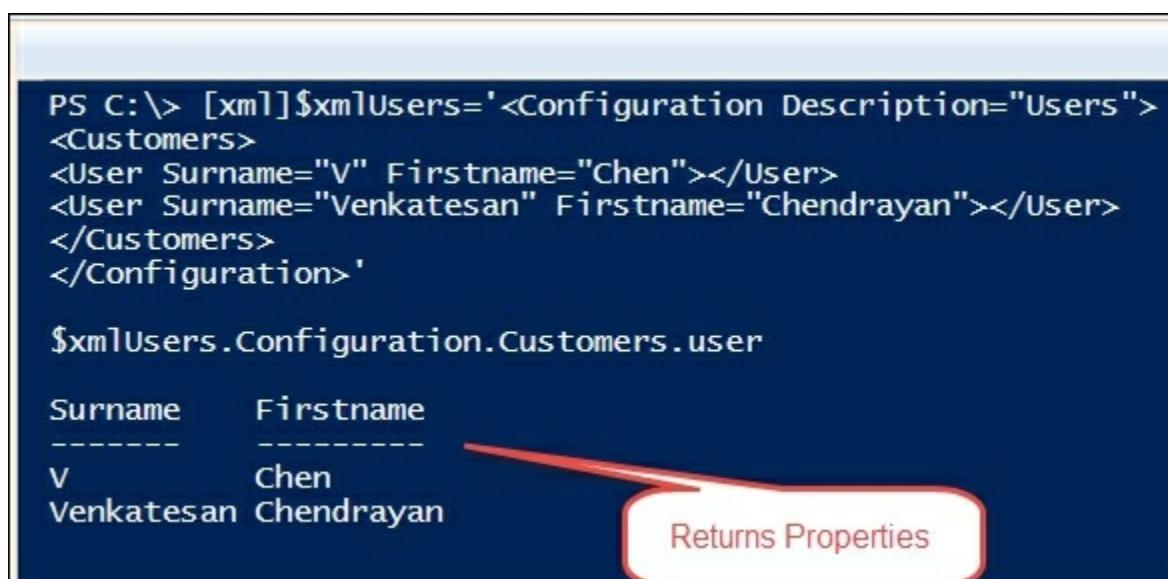
```
$o365Cred = Import-Clixml C:\Temp\SeccureString.xml
Import-Module MSOnline
$O365Session = New-PSSession -ConfigurationName
Microsoft.Exchange -ConnectionUri
https://outlook.office365.com/powershell-liveid/??
proxymethod=rps
-Credential $o365Cred -Authentication Basic -
AllowRedirection
Import-PSSession $O365Session
Connect-MsolService -Credential $o365Cred
```

As we passed the XML file as the value for the credential parameter, we don't need to enter the credential every time we connect to the Exchange Online session. Try to save the credential in a secured location.

It's really very handy and easy to create XML on the fly. For example, using the following code, we can create simple user XML:

```
[xml]$xmlUsers='<Configuration Description="Users">
<Customers>
<user surname="V" firstname="Chen"></user>
<user surname="Venkatesan" firstname="Chendrayan"></user>
</Customers>
</Configuration>'
$xmlUsers.Configuration.Customers.user
```

Take a look at the following image:



The screenshot shows a PowerShell session with the following command and output:

```
PS C:\> [xml]$xmlUsers='<Configuration Description="Users">
<Customers>
<User Surname="V" Firstname="Chen"></User>
<User Surname="Venkatesan" Firstname="Chendrayan"></User>
</Customers>
</Configuration>'

$xmlUsers.Configuration.Customers.user
```

Surname	Firstname
V	Chen
Venkatesan	Chendrayan

A red callout bubble points from the bottom right of the table towards the text "Returns Properties".

Here, we can see that `Surname` and `FirstName` are the properties of `User`. `User` is the property of `Customers` and `Customers` is the property of `Configuration`.

Exploring COM and Automation

Component Object Model (COM) is a binary interface used by developers to create reusable software components. The .NET framework and COM interfaces allow us to perform many admin tasks. `New-Object` is a magical command we use to create an instance of the COM object; similarly, we can create an instance of the Microsoft .NET Framework.

Windows PowerShell 5.0 includes a new COM implementation, which offers significant performance improvements.

Over the internet, we come across many COM Automation examples for Internet Explorer. So, in this exercise, we will explore MS Excel and MS Word Automation using PowerShell, which is also popular. IT managers need data in an Excel or Word format. Indeed, we can export to CSV and manipulate it as required. However, here, we need to exercise the usage of the COM object. So, let's use an Excel COM object for the same.

Perform the following steps:

1. Use `New-Object` to create an Excel COM object, this is a Program ID. Run the following command:

```
$Excel = New-Object -ComObject Excel.Application
```

2. Now, let's take a look at the events, methods, properties, and parameterized properties. Run the following command:

```
$Excel = New-Object -ComObject Excel.Application
"Excel Object has {0} events , {1} methods , {2} properties
and {3} parameterizedproperty" -f ($Excel | GM -MemberType
Event).Count ,
```

```
($Excel | GM -MemberType Method).Count , ($Excel | GM -MemberType Property).Count , ($Excel | GM -MemberType ParameterizedProperty).Count)
```

The output is illustrated in the following image:

A screenshot of a PowerShell window showing the output of a command. The command is: `$Excel = New-Object -ComObject Excel.Application`. The output is: "Excel Object has {0} events , {1} methods , {2} properties and {3} parameterizedproperty" -f ((`$Excel | GM -MemberType Event`).Count , (`$Excel | GM -MemberType Method`).Count , (`$Excel | GM -MemberType Property`).Count , (`$Excel | GM -MemberType ParameterizedProperty`).Count). Below this, it says "Excel Object has 43 events , 171 methods , 225 properties and 9 parameterizedproperty". Four red circles are overlaid on the output, labeled 1 through 4, pointing to the numbers 43, 171, 225, and 9 respectively.

The points marked in the figure are explained in the following list:

- 1: This shows that there are 43 events
- 2: This shows that there are 171 methods
- 3: This shows that there are 225 properties
- 4: This shows that there are 9 parameterized properties

3. Simply pipe and get members such as in the following command:

```
$Excel | GM
```

4. Invoke the `Speak` method from the `Speech` property. Run the following command:

```
$Excel = New-Object -ComObject Excel.Application  
$Excel.Speech.Speak("Opening Excel please wait!")
```

This will not open Excel for now, but we can listen to the audio—just for the purpose of this demo.

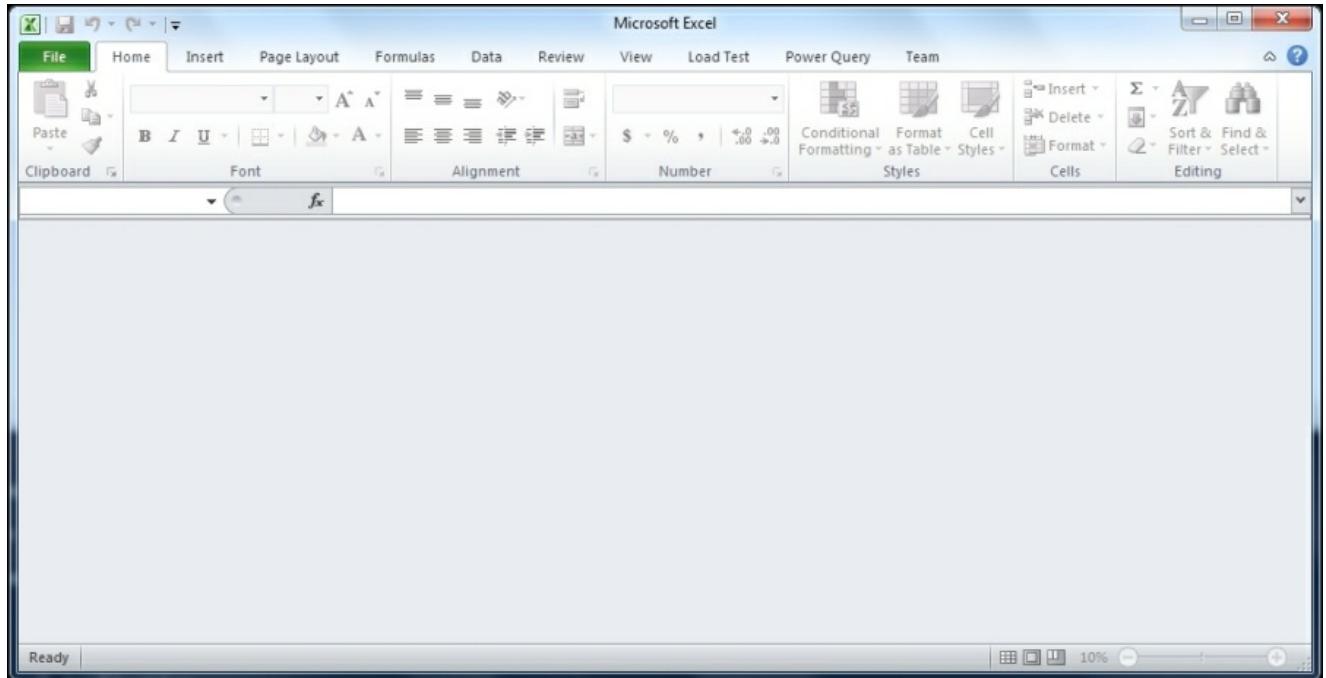
5. Now, set the `visible` property to `$true` to open Excel. Run the following command:

```
$Excel = New-Object -ComObject Excel.Application  
$Excel.Visible
```

This returns `False`. Run the following command:

```
$Excel = New-Object -ComObject Excel.Application  
$Excel.Visible = $true
```

This opens Excel in your localhost, as shown in the following screenshot:



Here, we opened Excel using the PowerShell `ComObject` property.

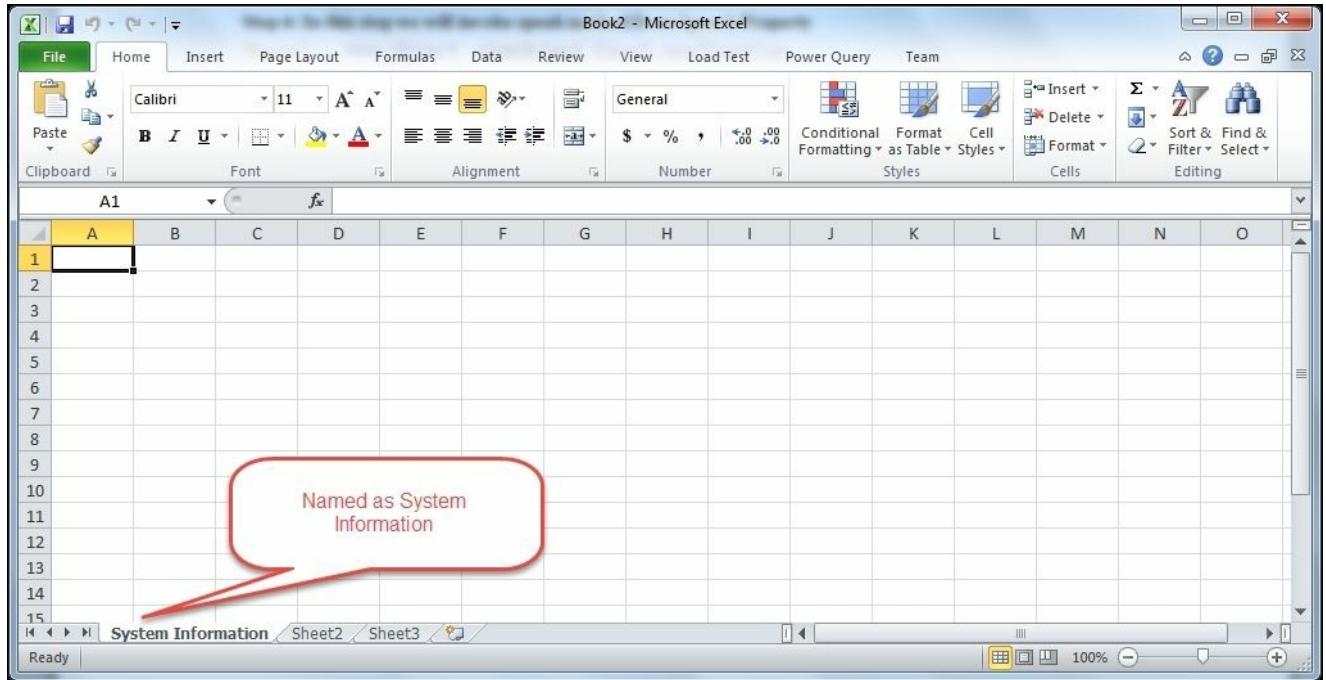
6. Now, let's add a workbook. Run the following command:

```
$Excel = New-Object -ComObject Excel.Application  
$Excel.Visible = $true  
$Wb = $Excel.Workbooks.Add()
```

7. After this, name the Work Sheet through the following command:

```
$Excel = New-Object -ComObject Excel.Application  
$Excel.Visible = $true  
$Wb = $Excel.Workbooks.Add()  
$worksheet = $Wb.Worksheets.Item(1)  
$worksheet.Name = "System Information"
```

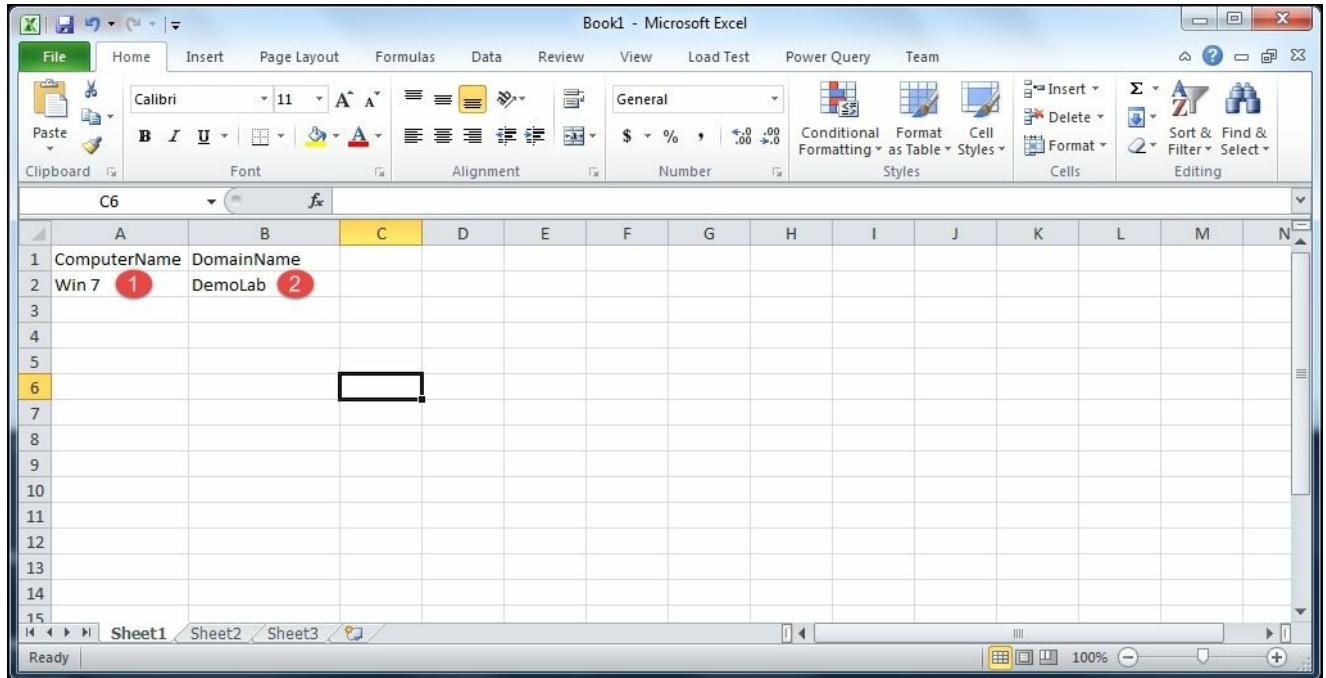
The output is illustrated in the following image:



8. Insert some values in the cells. Run the following command:

```
$Excel = New-Object -ComObject Excel.Application  
$Excel.Visible = $true  
$Wb = $Excel.Workbooks.Add()  
$worksheet = $Wb.Worksheets.Item(1)  
$worksheet.Cells.Item(1,1) = "ComputerName"  
$worksheet.Cells.Item(2,1) = $env:COMPUTERNAME  
$worksheet.Cells.Item(1,2) = "DomainName"  
$worksheet.Cells.Item(2,2) = $env:USERDOMAIN
```

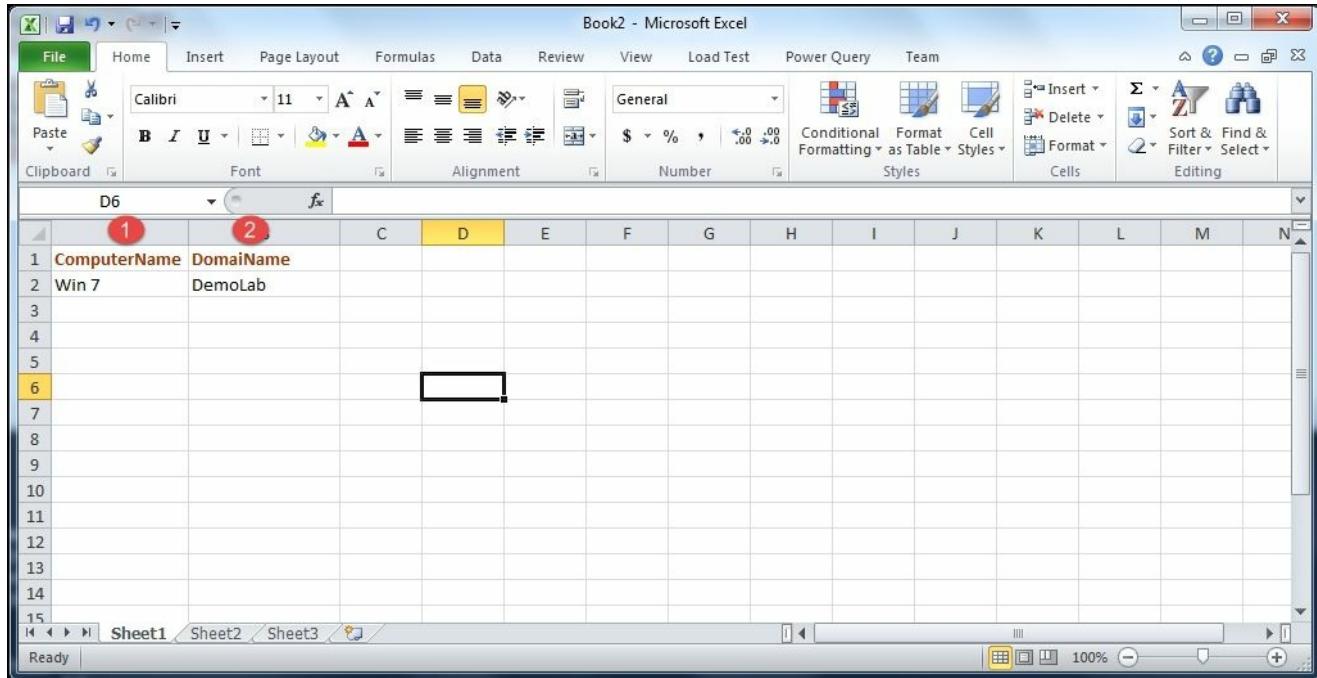
The output is illustrated in the following image:



Now that we have inserted some names for the localhost and domain name in the Excel sheet, let's change the color of this text. Run the following command:

```
$Excel = New-Object -ComObject Excel.Application  
$Excel.Visible = $true  
$Wb = $Excel.Workbooks.Add()  
$worksheet = $Wb.Worksheets.Item(1)  
$worksheet.Cells.Item(1,1) = "ComputerName"  
$worksheet.Cells.Item(1,1).Font.Bold = $true  
$worksheet.Cells.Item(1,1).Font.ColorIndex = 53  
$worksheet.Cells.Item(2,1) = $env:COMPUTERNAME  
$worksheet.Cells.Item(1,2) = "DomainName"  
$worksheet.Cells.Item(1,2).Font.Bold = $true  
$worksheet.Cells.Item(1,2).Font.ColorIndex = 53  
$worksheet.Cells.Item(2,2) = $env:USERDOMAIN
```

The output is illustrated in the following image:



The points marked in the figure are explained in the following list:

- 1: This shows that background color is set to index 53, which is brown color, and font weight is bold
- 2: This shows that same styles are applied as 1

9. Invoke the `SaveAs` method to save the Excel file through the following command:

```
$Excel = New-Object -ComObject Excel.Application
$Excel.Visible = $false
$Wb = $Excel.Workbooks.Add()
$worksheet = $Wb.Worksheets.Item(1)
$worksheet.Cells.Item(1,1) = "ComputerName"
$worksheet.Cells.Item(1,1).Font.Bold = $true
$worksheet.Cells.Item(1,1).Font.ColorIndex = 53
$worksheet.Cells.Item(2,1) = $env:COMPUTERNAME
$worksheet.Cells.Item(1,2) = "DomainName"
$worksheet.Cells.Item(1,2).Font.Bold = $true
$worksheet.Cells.Item(1,2).Font.ColorIndex = 53
$worksheet.Cells.Item(2,2) = $env:USERDOMAIN

$Wb.SaveAs ('C:\Temp\Output.xlsx')
```

In the preceding code, we have set the Excel visible property to false.

Using an Excel COM object, we can insert graphs, manipulate information, merge cells, and so on. Explore all the objects and create your code as required.

Now, let's explore the Microsoft Word object. This is similar to an Excel COM object exercise, so we will not cover each step. Instead, let's consider a sample code using the Word COM object. Always remember to release the COM object because it runs in the background. Before we begin, let's discuss the steps to release the COM object.

This is how the background chunk appears in your system:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
754	96	83284	101928	572	4.80	1832	WINWORD
749	74	69424	105144	555	1.78	3380	WINWORD
737	76	69704	106948	565	1.90	4664	WINWORD
740	76	66452	103424	565	1.83	6748	WINWORD
718	68	60880	82180	521	1.33	8148	WINWORD
853	112	122976	97560	714	1,085.77	11228	WINWORD
717	68	61188	82636	521	1.65	12024	WINWORD
714	68	61324	82872	521	2.11	12924	WINWORD
712	67	61084	55032	518	1.45	13108	WINWORD
740	76	69216	106416	562	2.31	13440	WINWORD
719	67	61312	55640	519	1.86	13884	WINWORD

Background
chunks!

Let's kill this process using the `Stop-Process` command. The result is illustrated in the following image:

```
PS C:\> Stop-Process -Name WINWORD -Verbose 1
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (1832)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (3380)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (4664)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (6748)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (8148)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (10044)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (12024)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (12924)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (13108)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (13440)".
VERBOSE: Performing the operation "Stop-Process" on target "WINWORD (13884)". 2
```

While exploring the members, I observed that it uses the `Quit` method,

so I used it to close Word. Take a look at the following image:

```
PS C:\> $word = New-Object -ComObject Word.Application  
$word.Quit() 1  
  
PS C:\> Get-Process -Name WINWORD 2  
Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName  
----- ----- ----- ----- ----- -----  
 742     68   61120    82808   521   1.36  3552 WINWORD  
 956    111   92980   150068   644   9.92  8220 WINWORD
```

Still not removed from chunk!

Here, we can see that nothing's turned up, and there are still two Word processes running. One of the processes is running in the background, and the other has opened up because I am writing this Module 2 of the course in MS Word.

With reference to the MSDN documentation, I tried to use the following method:

```
$word = New-Object -ComObject Word.Application  
[System.Runtime.InteropServices.Marshal]::ReleaseComObject($word)  
Remove-Variable -Name word -Verbose
```

As expected, it didn't turn up. The result is illustrated in the following image:

```
PS C:\> Stop-Process -Name WINWORD  
  
PS C:\> $word = New-Object -ComObject Word.Application  
[System.Runtime.InteropServices.Marshal]::ReleaseComObject($word)  
0 1  
  
PS C:\> Get-Process -Name WINWORD 2  
Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName  
----- ----- ----- ----- ----- -----  
 829      73   66844    83664   551   1.40  11368 WINWORD
```

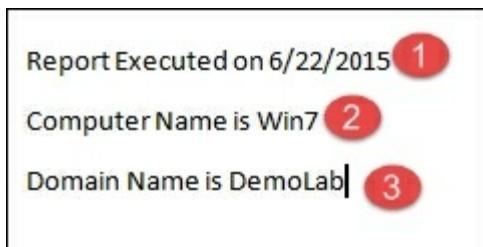
Finally, the following code worked as we expected it to:

```
$Word = New-Object -ComObject Word.Application
[System.Runtime.InteropServices.Marshal]::ReleaseComObject([System.ComObject]$Word) | out-null
[System.GC]::Collect()
[System.GC]::WaitForPendingFinalizers()
```

Now, let's try to create a Word document that has a report executed date, computer name, and domain name. Run the following command:

```
$word = New-Object -ComObject "Word.application"
$word.Visible = $true
$doc = $word.Documents.Add()
$doc.Activate()
$word.Selection.Font.Name = "Calibri"
$word.Selection.Font.Size = "10"
$word.Selection.TypeText("Report Executed on " + (Get-Date).ToString("MM/dd/yyyy"))
$word.Selection.TypeParagraph()
$word.Selection.TypeText("Computer Name is $env:computerName")
$word.Selection.TypeParagraph()
$word.Selection.TypeText("Domain Name is $env:USERDOMAIN")
$word.Selection.TypeParagraph()
$word.Quit()
```

The output of the preceding code output is shown in the following image:



The points marked in the figure are explained in the following list:

- 1: The date is converted to short date string format
- 2: The \$env:computerName variable is used to get the computer name
- 3: The \$env:USERDOMAIN variable is used to get the domain name

Let's create a small Word document that will retrieve information from active directory and build a Word document. The desired output will be similar to the following screenshot:



Venkatesan, C. (Chendrayan)

Department: ICT, Office Phone: +31xxxxxxxxx, Email: <Email Address>

Objective

Skills

Education

Projects

I hereby declare that the above written particulars are true to the best of my knowledge and belief.

Name : Venkatesan, C. (Chendrayan)

Date : 06/22/2015 14:45:12

Let's take a look at how this works. Perform the following steps:

1. Create a variable to save the username and filename as required.
Run the following command:

```
$FileName = $Env:USERNAME  
$savepath="C:\$FileName.docx"
```
2. Create an instance of a Word application using the Word.Application command. Add the document using the Add() property and select the document using the Selection property. Run the following command:

```
$word = New-Object -ComObject "Word.Application"
$doc=$word.documents.Add()
$Resume=$word.Selection
```

3. Set the style and use the `Get-ADUser` command to fetch the AD properties. Run the following command:

```
$Resume.Style="Title"
$UserName = Get-ADUser -Identity $ENV:USERNAME -Properties *
$Picture = Get-ADUser -Identity $ENV:USERNAME -Properties thumbnailphoto
```

4. Save the picture in the desired location with the following command:

```
$Picture.thumbnailphoto | Set-Content "C:\Photo.jpg" -
Encoding Byte
$Resume.TypeText("$(($UserName.Name))")
$Resume.InlineShapes.AddPicture("C:\Photo.jpg")
```

5. Retrieve the required information from AD using the following command:

```
$Resume.TypeParagraph()
$Resume.Style="Normal"
$Resume.TypeText("Department: $($UserName.Department), 
Office Phone: $($UserName.OfficePhone), Email:
 $($UserName.mail)")
```

6. Type out the text as required using the following code:

```
$Resume.TypeParagraph()
$Resume.Style="SubTitle"
$Resume.TypeText(("Objective"))
$Resume.TypeParagraph()
$Resume.Style="SubTitle"
$Resume.TypeText(("Skills"))
$Resume.TypeParagraph()
$Resume.Style="SubTitle"
$Resume.TypeText(("Education"))
$Resume.TypeParagraph()
$Resume.Style="SubTitle"
$Resume.TypeText(("Projects"))
$Resume.TypeParagraph()
$Resume.TypeParagraph()
$Resume.TypeParagraph()
```

```

$Resume.Style="SubTitle"
$Resume.TypeText(("I hereby declare that the above written
particulars are true to the best of my knowledge and
belief."))
$Resume.TypeParagraph()
$Resume.TypeParagraph()
$Resume.TypeParagraph()
$Resume.Style="Strong"
$Resume.TypeText("Name : $($UserName.Name)")
$Resume.TypeParagraph()
$Date = Get-Date
$Resume.TypeText("Date : $($Date)")

```

- Save the file. Run the following command:

```

$doc.SaveAs ([ref]$savepath)
$doc.Close()

```

- Run the following command to quit Word and open the file:

```

$word.quit()
Invoke-Item $savepath
You can use the complete code, which is as follows:
$fileName = $Env:USERNAME
$savepath="C:\$fileName.docx"
$word = New-object -ComObject "Word.Application"
$doc=$word.documents.Add()
$Resume=$word.Selection
$Resume.Style="Title"
$UserName = Get-ADUser -Identity $ENV:USERNAME -Properties *
$Picture = Get-ADUser -Identity $ENV:USERNAME -Properties
thumbnailphoto
$Picture.thumbnailphoto | Set-Content "C:\Photo.jpg" -
Encoding Byte
$Resume.TypeText("$($UserName.Name)")
$Resume.InlineShapes.AddPicture("C:\Photo.jpg")
$Resume.TypeParagraph()
$Resume.Style="Normal"
$Resume.TypeText("Department: $($UserName.Department),
Office Phone: $($UserName.OfficePhone), Email:
 $($UserName.mail)")
$Resume.TypeParagraph()
$Resume.Style="SubTitle"
$Resume.TypeText(("Objective"))
$Resume.TypeParagraph()
$Resume.Style="SubTitle"

```

```
$Resume.TypeText("Skills")
$Resume.TypeParagraph()
$Resume.Style="SubTitle"
$Resume.TypeText("Education")
$Resume.TypeParagraph()
$Resume.Style="SubTitle"
$Resume.TypeText("Projects")
$Resume.TypeParagraph()
$Resume.TypeParagraph()
$Resume.TypeParagraph()
$Resume.Style="SubTitle"
$Resume.TypeText("I hereby declare that the above written
particulars are true to the best of my knowledge and
belief.")
$Resume.TypeParagraph()
$Resume.TypeParagraph()
$Resume.TypeParagraph()
$Resume.Style="Strong"
$Resume.TypeText("Name      :      $($UserName.Name)")
$Resume.TypeParagraph()
$Date = Get-Date
$Resume.TypeText("Date      :      $($Date)")
$doc.SaveAs([ref]$savepath)
$doc.Close()
$word.quit()
Invoke-Item $savepath
```

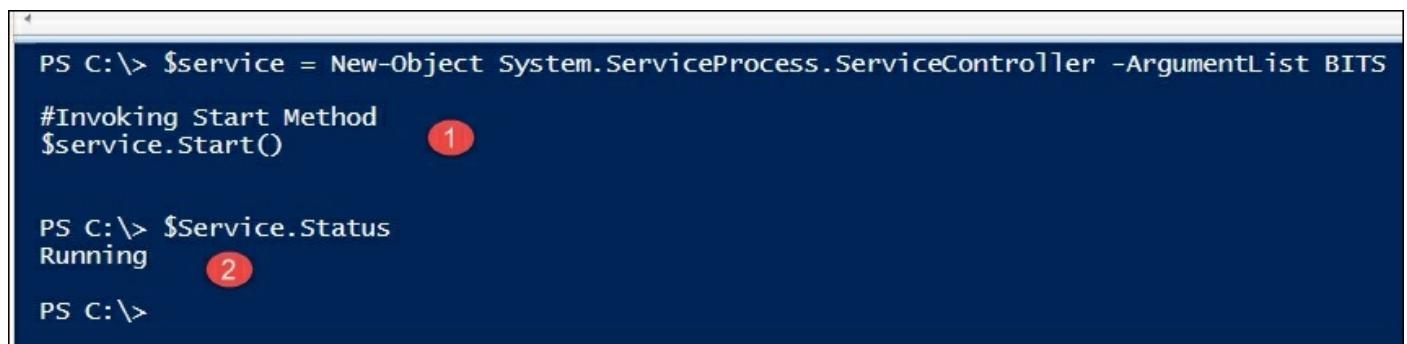
Exploring .NET objects

An object is nothing but a combination of methods and properties in PowerShell. Using Windows PowerShell, we can store a reference to an object to a variable and use it in the current shell as required.

We discussed in [Chapter 1, Getting Started with Windows PowerShell](#) that PowerShell takes advantage of the underlying .NET framework. So, the objects are a representation of the parts and actions to use it. An object is a combination of the properties and methods (*Objects = Properties + Methods*).

For example, a Windows service object has properties and methods. The properties are `Get` and `Set`, and the methods are invoked to perform a meaningful operation.

Consider the following image:



The screenshot shows a Windows PowerShell session with the following commands and output:

```
PS C:\> $service = New-Object System.ServiceProcess.ServiceController -ArgumentList BITS
#Invoking Start Method
$service.Start() ①

PS C:\> $Service.Status
Running ②

PS C:\>
```

Annotations with red circles and numbers:

- Annotation 1: A red circle with the number "1" is placed next to the line `$service.Start()`.
- Annotation 2: A red circle with the number "2" is placed next to the line `Running`.

Creating .NET objects

We have discussed objects in a few of the preceding examples as well. In this section, we will explore the .NET objects in PowerShell. In order to create a .NET object, we will use the same `New-Object` cmdlet, but instead of the COM object, we will use different types of objects.

To create a .NET object for the system version, we will use the following code:

```
$Version = New-Object -TypeName System.Version -ArgumentList  
1.2.3.4  
$Version
```

The output is illustrated in the following image:

PS C:\> \$Version = New-Object -TypeName System.Version -ArgumentList 1.2.3.4
\$Version ① ② ③ ④ ⑤ ⑥
Major Minor Build Revision
--- --- --- ---
1 2 3 4
PS C:\> | Output

The points marked in the figure are explained in the following list:

- 1: Here, we created a variable \$Version
- 2: Here, we used the New-Object cmdlet
- 3: Here, we are using the parameter -TypeName to call the .NET framework class
- 4: Here, we are calling the System.Version .NET framework class
- 5: Here, we are using -ArgumentList to pass values to the constructor of the class
- 6: The passed values are 1.2.3.4.

As we have discussed a little about typecasting in the previous topics, we can do the same here as well. Run the following command:

```
[System.Version]"1.2.3.4"
```

The result is the same, as shown in the following image:

The diagram shows a PowerShell window with the following command and output:

```
PS C:\> $Version = New-Object -TypeName System.Version -ArgumentList 1.2.3.4
$Version
```

The output is a table:

Major	Minor	Build	Revision
1	2	3	4

A red callout bubble labeled "Output" points to the table.

Annotations (red circles with numbers):

- 1: Points to the variable name \$Version.
- 2: Points to the command New-Object.
- 3: Points to the argument 1.2.3.4.
- 4: Points to the table header.
- 5: Points to the table body.
- 6: Points to the final closing bracket of the command.

Extending .NET objects for Administrations and Development tasks

Using PowerShell, we can extend the instance of the .NET object. To do this, we will use the `Add-member` cmdlet. In the following example, we will discuss extending the `System.String` class:

```
$string = New-Object -TypeName System.String -ArgumentList
"PowerShell Rocks!"
($string) .GetType()
```

The output of the code we just saw is illustrated in the following image:

The diagram shows a PowerShell window with the following command and output:

```
PS C:\> $string = New-Object -TypeName System.String -ArgumentList "PowerShell Rocks!"
($string).GetType()
```

The output is a table:

IsPublic	IsSerial	Name	BaseType
True	True	String	System.Object

Annotations (red circles with numbers):

- 1: Points to the method `GetType()`.
- 2: Points to the output table.
- 3: Points to the column headers.
- 4: Points to the value "System.Object" under the BaseType column.

The points marked in the figure are explained in the following list:

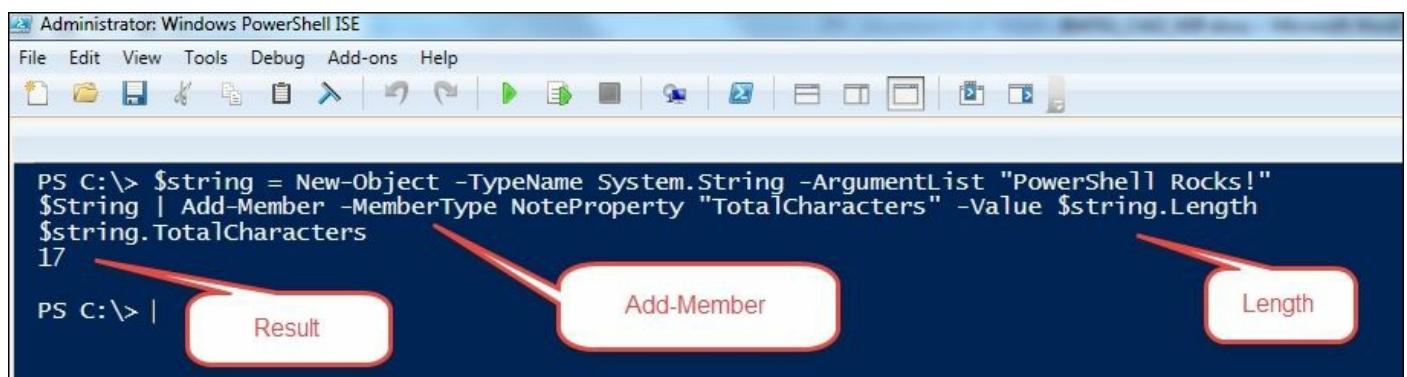
- 1: In the above code, we use `GetType()` method to explore the type name. It's a string.

- **2:** Here, the argument passed for the constructor of System.String class is PowerShell Rocks !
 - **3:** Here, the value returned is String
 - **4:** The base objects in System.Object (Note: If we do \$var = 123 ; \$var.GetType () the base objects returns System.Value). In the above code we have created a String object. So, the base type shows the ultimate base class of all classes in the .NET Framework

Now, let's add `NoteProperty`. This will return the total character counts from the value passed through, which is `ArgumentList`. Run the following command:

```
$string = New-Object -TypeName System.String -ArgumentList  
"PowerShell Rocks!"  
$String | Add-Member -MemberType NoteProperty "TotalCharacters"  
-Value $string.Length  
$string.TotalCharacters
```

The output is illustrated in the following image:



Let's consider another example where we concatenate the `ScriptMethod` object to a string object. Run the following command:

```
$string = New-Object -TypeName System.String -ArgumentList  
"PowerShell Rocks!"  
$String | Add-Member -MemberType ScriptMethod -Name  
"ConcatCustom" -Value {[String]::Concat($this, " Demo")}  
$string.ConcatCustom()
```

The preceding code appended the `Demo` string to the `ArgumentList` string,

PowerShell Rocks!.

The output is illustrated in the following image:

A screenshot of a Windows PowerShell window. The command entered is:

```
PS C:\> $string = New-Object -TypeName System.String -ArgumentList "PowerShell Rocks!"  
$String | Add-Member -MemberType ScriptMethod -Name "ConcatCustom" -Value {[String]::Concat($this, " Demo")}  
$string.ConcatCustom()  
PowerShell Rocks! Demo
```

Red numbered circles point to specific parts of the command:

- 1: Points to the `-MemberType` parameter.
- 2: Points to the `ConcatCustom` method name.
- 3: Points to the `[String]::Concat` value assignment.
- 4: Points to the `$this` automatic variable.
- 5: Points to the first parameter of the `Concat` method.
- 6: Points to the second parameter of the `Concat` method.
- 7: Points to the string `" Demo"`.
- 8: Points to the opening parenthesis of the `ConcatCustom` call.
- 9: Points to the closing parenthesis of the `ConcatCustom` call.

The points marked in the figure are explained in the following list:

- 1: This shows that the member type of the `Add-Member` cmdlet is `ScriptMethod`
- 2: Here, I named the script as `CustomConcat` just for demo purposes — You can choose any custom name
- 3: Here, the `value` parameter defines the script
- 4: Here, we used the `[String]::Concat` method just for simple demo purposes
- 5: Here, we see that the `Concat` method accepts two string parameters (`string1, string2`)
- 6: Here, we can see that the `value` parameter accepts the `$this` automatic variable as well
- 7: Here, we see that `Demo` is a string we need to concatenate
- 8: Here, we invoke our script method
- 9: Here, we see the output as `PowerShell Rocks! Demo`

Extending the .NET Framework types

Using Windows PowerShell, we can define a Windows .NET Framework class in the session and instantiate the object using `New-Object`.

PowerShell allows us to use the inline C# code using a **here-string**. To define a here-string in Windows PowerShell, we should use `@"` and close it with `"@`—anything between these marks is a here-string, as shown in the following snippet:

```
@"
```

```
"here strings"
```

```
"@
```

From Windows PowerShell 5.0 onward, we can write the class in Windows PowerShell. The `Class` keyword is supported only in PowerShell version 5.0. For now, let's take a look at how the `Add-Type` cmdlet works. In the following example, we will take a look at code with the inline C# code:

```
$sourcecode = @"
public class Calculator
{
    public static int Add(int a, int b)
    {
        return (a + b);
    }
}
"@
```

```
Add-Type -TypeDefinition $sourcecode
[Calculator]
```

The output is illustrated in the following image:

```
PS C:\> Add-Type -TypeDefinition $sourcecode
[Calculator]

IsPublic IsSerial Name
----- ----- -----
True     False    Calculator

PS C:\> |
```

Now, we can call the `Add` method directly without using the `New-Object` cmdlet. Run the following command:

```
$sourcecode = @"
public class Calculator
{
    public static int Add(int a, int b)
    {
        return (a + b);
}
```

```

}
"@"
Add-Type -TypeDefinition $sourcecode
[Calculator]::Add(10,5)

```

PowerShell allows us to execute the static methods of a class directly in PowerShell. `System.Math` has a static method called `sqrt`, which can be used as shown in the following command:

```
[System.Math]::Sqrt(4)
```

In the following example, we will create a simple class using the `Class` keyword:

```

Class Demo
{
    #Properties
    [string]$FirstName = "Chen"
    [string]$surname = "V"
    #Methods - This will print the values and just for demo
    [string]GetInformation() {
        return "$($this.FirstName) $($this.surname)"
    }
}

```

Following is the alternate method to instantiate the class:

- We have used the keyword `Class` and named it as `Demo`
- `$FirstName` and `$surname` are properties
- `GetInformation` is a method and `$this` is an automatic variable used
- Used `$var` variable and instantiating the class using `New-Object` cmdlet. With the same code the class can instantiated using `$var = [Demo]::new(); $var.GetInformation()`

The output is shown in the following image:

```

1 $var = New-Object -TypeName Demo
2 $var
3
4
FirstName surname
-----
Chen 5 V 6

```

Let's explore the property, as shown in the following image:

```

PS C:\> $var | GM

TypeName: Demo

Name      MemberType Definition
----      -----
Equals    Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetInformation Method   void GetInformation()
GetType    Method     type GetType()
ToString   Method     string ToString()
FirstName  Property   System.Object FirstName {get;set;}
surname   Property   System.Object surname {get;set;}

```

The class `Demo` we created in the preceding section has properties and methods, but we practically don't need the `GetInformation` method because it just prints the property values. Let's create another class with the `SetInformation` method which updates the property values. Run the following command:

```

Class Demo
{
    #Properties
    [string]$FirstName = "Chen"
    [string]$surname = "V"
    #Methods - This will print the values and just for demo
    GetInformation() {
        $this.FirstName
        $this.surname
    }
}

```

```

}
SetInformation([string]$fn,[string]$sn) {
    $this.FirstName = $fn
    $this.surname = $sn
}
$var = New-Object -TypeName Demo
$var.SetInformation("Chendrayan" , "Venkatesan")
$var

```

The preceding code returns the following output:

```

FirstName   surname
-----   -----
Chendrayan Venkatesan

```

We will cover more information about the classes in the *Exploring Windows PowerShell 5.0* section.

Building advanced scripts and modules

Windows PowerShell scripting is used imperatively to perform complex operations using cmdlets. Windows PowerShell supports variables, functions, branching, loops, structured exception handling and, as we all know, .NET integration.

The PowerShell script file has a PS1 extension. For example, let's save the following file as PS1 and execute it:

```

Function Get-Information
{
    (Get-Service) | Where({$_.Status -eq 'Stopped'})
}
Get-Information

```

The preceding script is a very basic code to retrieve all the running services. Consider the following image:

Status	Name	DisplayName
Stopped	AdobeFlashPlaye...	Adobe Flash Player Update Service
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Stopped	AppIDSvc	Application Identity
Stopped	aspnet_state	ASP.NET State Service
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Stopped	BDESVC	BitLocker Drive Encryption Service
Stopped	Blackberry Devi...	Blackberry Device Manager
Stopped	Browser	Computer Browser
Stopped	c2wts	Claims to Windows Token Service
Stopped	clr_optimizatio...	Microsoft .NET Framework NGEN v2.0....
Stopped	clr_optimizatio...	Microsoft .NET Framework NGEN v2.0....
Stopped	clr_optimizatio...	Microsoft .NET Framework NGEN v4.0....
Stopped	clr_optimizatio...	Microsoft .NET Framework NGEN v4.0....
Stopped	COMSysApp	COM+ System Application
Stopped	cphs	Intel(R) Content Protection HECI Se...
Stopped	defragsvc	Disk Defragmenter
Stopped	dot3svc	Wired AutoConfig
Stopped	ehRecvr	Windows Media Center Receiver Service
Stopped	ehSched	Windows Media Center Scheduler Service

Before taking a look at the advanced functions, let's consider the fundamentals of PowerShell scripting.

A PS1 file is nothing but a text file that contains a sequence of PowerShell cmdlets to perform certain complex tasks. Before you execute a PowerShell script, ensure that the script execution policy is set as required. There are different types of execution policies; to know them, simply execute the following help cmdlet:

```
help about_Execution_Policies -Detailed
```

Before creating PowerShell scripts, ensure that you plan and document the synopsis. Using the `Measure-Command` cmdlet, ensure that you optimize the performance of the code as required.

Now, let's take a look at the advanced function and save it as PS1 file. For this demo, I have used the PowerShell ISE, which has a snippet for advanced functions and advanced function complete.

The structure of the advanced function is shown in the following image:

```

1  <#...#> 1
11 function Verb-Noun 3
12 { 2
13     [CmdletBinding()]
14     [Alias()]
15     [OutputType([int])]
16     Param
17     (...) 5
18
19     Begin 6
20     {...}
21     Process 7
22     {...}
23     End 8
24     {...}
25 }
26
27
28
29
30
31
32
33
34
35
36
37
38 }

```

The points marked in the figure are explained in the following list:

- **1:** This is a comment block
- **2:** Using synopsis, description, examples, and notes makes the script readable and easy to understand
- **3:** Name of the function—always a Verb-Noun Combination is the best practice
- **4:** The `CmdletBinding` attribute is an attribute of functions, which makes a function work similarly to a compiled cmdlet that is written in C#
- **5:** Here, the param block is used to define parameters
- **6:** Here, the begin block is used to provide optional onetime pre-processing
- **7:** Here, the process block is used to provide record by record processing
- **8:** Here, the end block is used to provide optional one time post-processing

Let's take the example of a PowerShell script that retrieves system information and saves the output as an HTML file, which uses the CSS style sheet. Run the following code:

```
<#
```

```
.SYNOPSIS
Windows Machine Inventory Using PowerShell.

.DESCRIPTION
This script is to document the Windows machine. This script
will work only for Local Machine.

.EXAMPLE
PS C:\> .\System_Inventory.ps1

.OUTPUTS
HTML File OutPut ReportDate , General Information , BIOS
Information etc.

#>
#Set-ExecutionPolicy RemoteSigned -ErrorAction SilentlyContinue
$UserName = (Get-Item env:\username).Value
$ComputerName = (Get-Item env:\Computername).Value
$filepath = (Get-ChildItem env:\userprofile).value
Add-Content "$filepath\style.CSS" -Value " body {
    font-family:Calibri;
    font-size:10pt;
}
th {
    background-color:black;
    color:white;
}
td {
    background-color:#19ffff0;
    color:black;
}"
Write-Host "CSS File Created Successfully... Executing
Inventory Report!!! Please Wait !!!" -ForegroundColor Yellow
#ReportDate
$ReportDate = Get-Date | Select -Property DateTime | ConvertTo-
Html -Fragment
#General Information
$ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem |
Select -Property Model , Manufacturer , Description ,
PrimaryOwnerName , SystemType |ConvertTo-HTML -Fragment
#Boot Configuration
$BootConfiguration = Get-WmiObject -Class
Win32_BootConfiguration |
Select -Property Name , ConfigurationPath | ConvertTo-HTML -
Fragment
#BIOS Information
$BIOS = Get-WmiObject -Class Win32_BIOS | Select -Property
PSCoMputerName , Manufacturer , Version | ConvertTo-HTML -
Fragment
#Operating System Information
```

```

$OS = Get-WmiObject -Class Win32_OperatingSystem | Select -
Property Caption , CSDVersion , OSArchitecture , OSLanguage |
ConvertTo-Html -Fragment
#Time Zone Information
$TimeZone = Get-WmiObject -Class Win32_TimeZone | Select
Caption , StandardName |
ConvertTo-Html -Fragment
#Logical Disk Information
$Disk = Get-WmiObject -Class Win32_LogicalDisk -Filter
DriveType=3 |
Select SystemName , DeviceID , @{Name="size(GB)" ;Expression={"0:N1"} -f ($_.size/1gb)} , @{Name="freespace(GB)" ;Expression={"0:N1"} -f ($_.freespace/1gb)}} |
ConvertTo-Html -Fragment
#CPU Information
$SystemProcessor = Get-WmiObject -Class Win32_Processor |
Select SystemName , Name , MaxClockSpeed , Manufacturer ,
status |ConvertTo-Html -Fragment
#Memory Information
$PhysicalMemory = Get-WmiObject -Class Win32_PhysicalMemory |
Select -Property Tag , SerialNumber , PartNumber , Manufacturer ,
DeviceLocator , @{Name="Capacity(GB)" ;Expression={"0:N1"} -f
($_.Capacity/1GB)}} | ConvertTo-Html -Fragment
#Software Inventory
$Software = Get-WmiObject -Class Win32_Product |
Select Name , Vendor , Version , Caption | ConvertTo-Html -
Fragment
ConvertTo-Html -Body "<font color = blue><H4><B>Report Executed
On</B></H4></font>$ReportDate
<font color = blue><H4><B>General Information</B></H4>
</font>$ComputerSystem
<font color = blue><H4><B>Boot Configuration</B></H4>
</font>$BootConfiguration
<font color = blue><H4><B>BIOS Information</B></H4></font>$BIOS
<font color = blue><H4><B>Operating System Information</B></H4>
</font>$OS
<font color = blue><H4><B>Time Zone Information</B></H4>
</font>$TimeZone
<font color = blue><H4><B>Disk Information</B></H4></font>$Disk
<font color = blue><H4><B>Processor Information</B></H4>
</font>$SystemProcessor
<font color = blue><H4><B>Memory Information</B></H4>
</font>$PhysicalMemory
<font color = blue><H4><B>Software Inventory</B></H4>
</font>$Software" -CssUri "$filepath\style.CSS" -Title "Server
Inventory" | Out-File "$FilePath\$ComputerName.html"

```

```
Write-Host "Script Execution Completed" -ForegroundColor Yellow  
Invoke-Item -Path "$FilePath\$ComputerName.html"
```

Note that the preceding code is not a function—it's just a script. Save this as PS1 and execute the script.

The sample output is shown in the following image:

Report Executed On				
*				
Tuesday, June 23, 2015 8:31:16 AM				
General Information				
Model	Manufacturer	Description	PrimaryOwnerName	SystemType
LIFEBOOK S761	FUJITSU	AT/AT COMPATIBLE	Windows User	x64-based PC
Boot Configuration				
Name	ConfigurationPath			
BootConfiguration	C:\windows			

The script has the following elements:

- Comment block
- Parameter declared for computer name and file path
- Creates CSS file in C:\ drive for styling
- Report executed date
- General system information
- Boot configuration information
- BIOS information
- OS information
- Time zone settings
- Logical disk information
- CPU information
- Memory information
- Software inventory
- Converts to HTML and outputs in HTML file
- Invoke the HTML file in default browser

Consider the following image:

```
1 #<#...#> 1
15
16 #region - param ... 2
21
22 #region - Create CSS... 3
37 Write-Host "CSS File Created Successfully... Executing Inventory Report!!! Please Wait !!!" -ForegroundColor Yellow
38 #region ReportDate ... 4
41
42 #region General Information ... 5
46
47 #region Boot Configuration ... 6
51
52 #region BIOS Information ... 7
55
56 #region Operating System Information ... 8
59
60 #region Time Zone Information ... 9
64
65 #region Logical Disk Information ... 10
70
71 #region CPU Information ... 11
75
76 #region Memory Information ... 12
80
81 #region Software Inventory ... 13
85
86 #region HTML Body ... 14
98
99 #region Invoke the HTML file... 15
```

Let's take an example where we will create an advanced function. Advanced functions include help, parameters, accepting pipelines, and so on.

Note

Note that, to do this easily, you can open PowerShell ISE, press *CTRL + J*, and choose the **Cmdlet (advanced function)**.

Perform the following steps:

1. Write a simple help block for a function:

```
<#
.Synopsis
    A script to retrieve OS information.
.DESCRIPTION
    This PowerShell script will retrieve OS information like
Name , OS Architecture, Serial Number and Last Bootup time.
    This script use CIM instance.
.EXAMPLE
    Get-OSInformation -ComputerName localhost
.EXAMPLE
    Get-OSInformation -ComputerName localhost ,
remotecomputer
```

.EXAMPLE

```
localhost , remotecomputer | Get-OSInformation  
#>
```

The help block has synopsis, description, and three examples.

2. Create a function using the keyword `function` and name it in the Verb-Noun form. In our example, this is `Get-OSInformation`, as shown in the following command:

```
function Get-OSInformation { #Code goes here}
```

3. Use `CmdletBinding`. Take a look at the following code:

```
[CmdletBinding(ConfirmImpact = 'Low', HelpUri =  
'http://chen.about-powershell.com')]
```

I have set the `ConfirmImpact` to low, and I have used `HelpUri` from my blog post—just for this example. You can use any valid site that has some information. You may wonder, why do we use `CmdletBinding`? It is to ease our function creation and to make additional validation for our parameters. For more information execute `help about_Functions_CmdletBindingAttribute`

4. Declare the parameters. In this example, we will use a single parameter, as shown in the following code:

```
Param  
(  
    # Param1 help description  
    [Parameter(Mandatory=$true,  
              HelpMessage = "Please enter valid host name",  
              ValueFromPipelineByPropertyName=$true,  
              ValueFromPipeline = $true,  
              Position=0)]  
    [String[]]$ComputerName  
)
```

Use the following code:

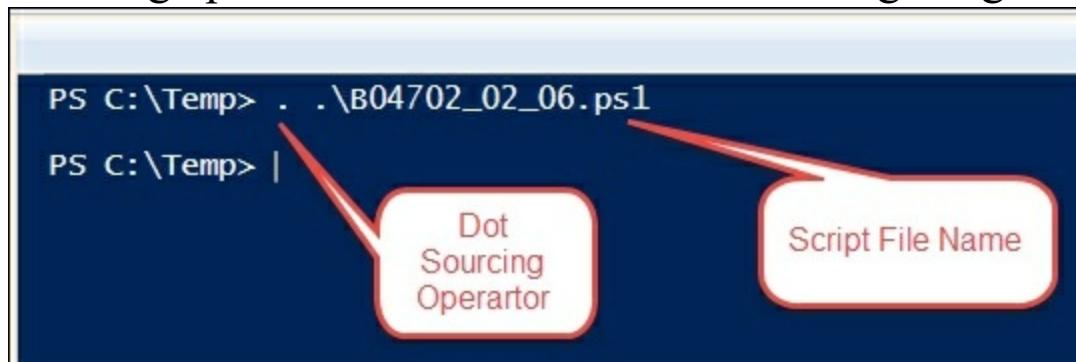
```
Begin  
{  
    #Intentionally left blank  
}  
Process
```

```

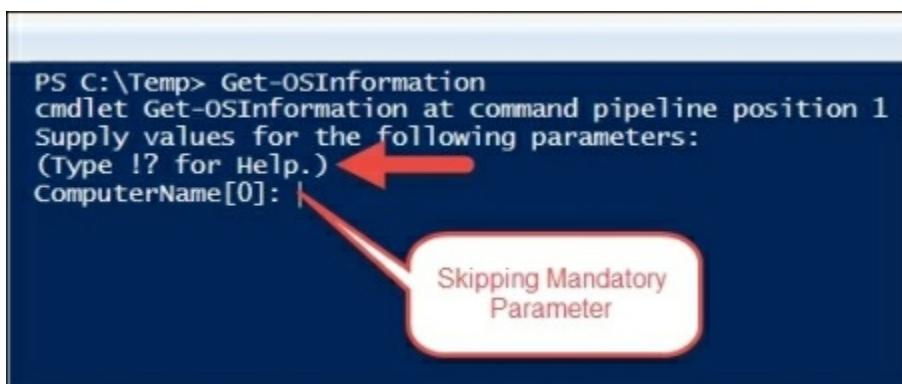
{
    foreach($computer in $ComputerName)
    {
        $params = @{'ComputerName' = $Computer
                    'Class' = 'CIM_OperatingSystem'}
        Get-CimInstance @Params |
            Select Caption , OSArchitecture , SerialNumber
        , LastBootUptime
    }
}
End
{
    #Intentionally left blank
}

```

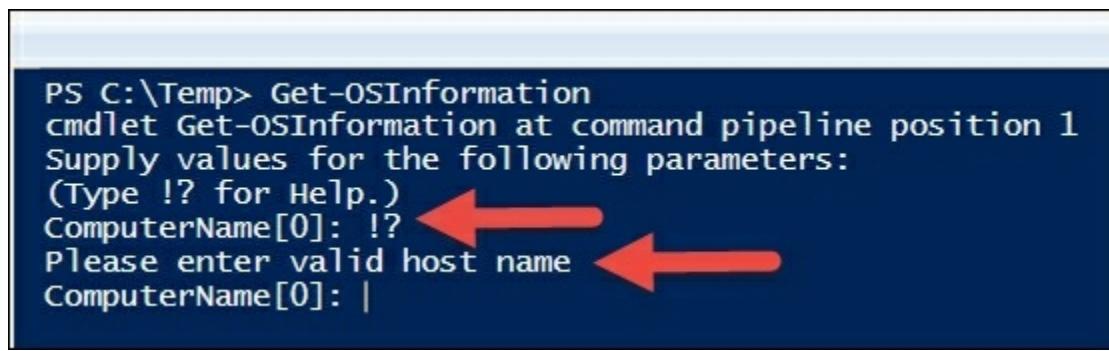
- Save the PS1 file in the desired location and call it using a dot sourcing operator. Take a look at the following image:



Now, let's call the function using `Get-OSInformation` by skipping the mandatory parameter, as shown in the following image:



Type !? to see the help text, as shown in the following image:



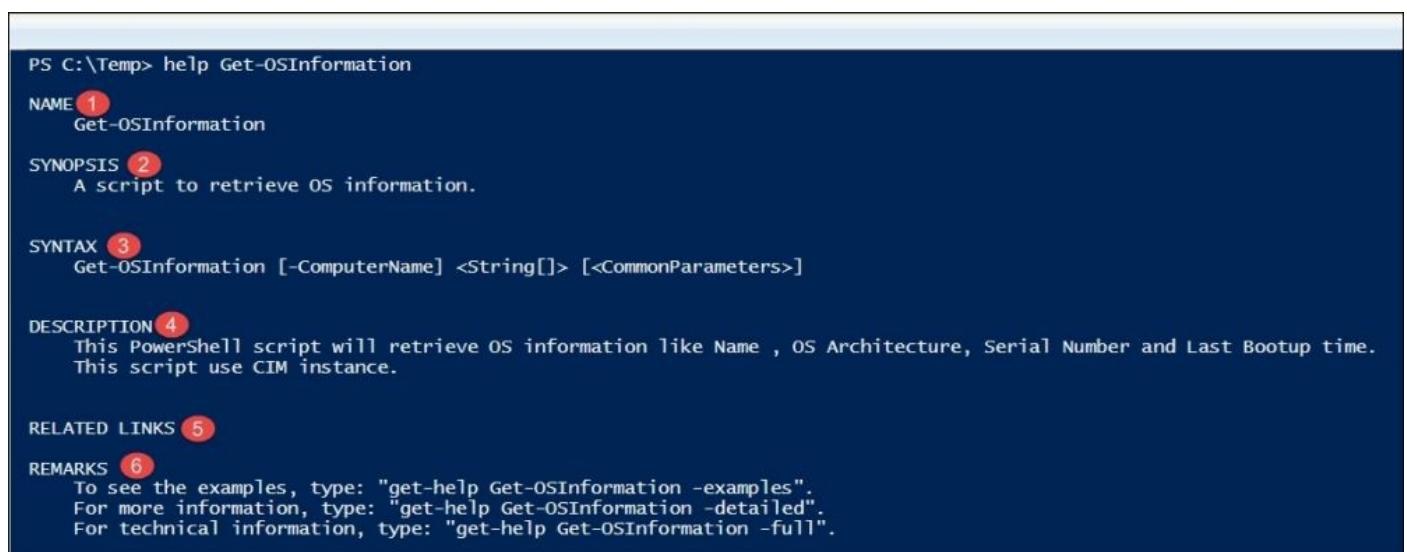
```
PS C:\Temp> Get-OSInformation
cmdlet Get-OSInformation at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
ComputerName[0]: !?
Please enter valid host name ←
ComputerName[0]: |
```

The screenshot shows a PowerShell window. The command 'Get-OSInformation' is run. A parameter block is displayed, asking for 'ComputerName[0]'. The placeholder 'Please enter valid host name' is highlighted with a red arrow. The cursor is at the end of the input line.

That's cool! Now, we can see the help message in the parameter block.

Using the help command, we can obtain more information about the command.

The output is illustrated in the following image:



```
PS C:\Temp> help Get-OSInformation
NAME ①
    Get-OSInformation

SYNOPSIS ②
    A script to retrieve OS information.

SYNTAX ③
    Get-OSInformation [-ComputerName] <String[]> [<CommonParameters>]

DESCRIPTION ④
    This PowerShell script will retrieve OS information like Name , OS Architecture, Serial Number and Last Bootup time.
    This script use CIM instance.

RELATED LINKS ⑤

REMARKS ⑥
    To see the examples, type: "get-help Get-OSInformation -examples".
    For more information, type: "get-help Get-OSInformation -detailed".
    For technical information, type: "get-help Get-OSInformation -full".
```

The screenshot shows the help output for 'Get-OSInformation'. Points are numbered 1 through 6: 1 points to the command name 'Get-OSInformation'; 2 points to the synopsis 'A script to retrieve OS information.'; 3 points to the syntax line 'Get-OSInformation [-ComputerName] <String[]> [<CommonParameters>]'; 4 points to the description block; 5 points to the 'RELATED LINKS' section; 6 points to the 'REMARKS' section.

The points marked in the figure are explained in the following list:

- 1: Name of the command appears here.
- 2: Whatever we enter in the synopsis appears here.
- 3: Here, syntax appears automatically and it shows the datatype as well.
- 4: Description is again customized—what we entered in the

description section appears here.

- 5: RELATED LINKS—appears by default if you have included .LINK in the comment block. Used for external web site links.
- 6: REMARKS—appears by default.

Note that here, the `help` command will not show the input and output parameters, and so on.

Use the `help Get-OSInformation -Full` command to explore more.

We used the blog post URL in the help message URI in cmdlet binding; so, to make use of this, we can use the following code:

```
help Get-OSInformation -Online
```

The complete code is available here:

```
<#
.Synopsis
    A script to retrieve OS information.
.DESCRIPTION
    This PowerShell script will retrieve OS information like Name
, OS Architecture, Serial Number and Last Bootup time.
    This script use CIM instance.
.EXAMPLE
    Get-OSInformation -ComputerName localhost
.EXAMPLE
    Get-OSInformation -ComputerName localhost , remotecomputer
.EXAMPLE
    localhost , remotecomputer | Get-OSInformation
#>
function Get-OSInformation {
    [CmdletBinding(ConfirmImpact = 'Low', HelpUri =
'http://chen.about-powershell.com')]
Param(
# Param1 help description
[Parameter(Mandatory=$true,
    HelpMessage = "Please enter valid host name",
    ValueFromPipelineByPropertyName=$true,
    ValueFromPipeline = $true,
    Position=0)]
    [String[]]$ComputerName)
Begin{<#Intentionally left blank#>}
```

```

Process
{
    foreach ($computer in $ComputerName)
    {
        $params = @{'ComputerName' = $Computer
                    'Class' = 'CIM_OperatingSystem'}
        Get-CimInstance @Params |
            Select Caption , OSArchitecture , SerialNumber ,
LastBootUptime
    }
}
End
{
    #Intentionally left blank
}
}

```

In this section, we will explore Windows PowerShell modules.

Windows PowerShell modules are packages that contain the PowerShell commands. Using modules, we can organize our commands and share them with others.

In this example, we will create a simple module as a demo and this will be a script module. We will discuss all types of modules and demos in the *Understanding PowerShell Modules* section.

A script module is a file (.psm1) that contains valid Windows PowerShell code. Script developers and administrators can use this type of module to create modules whose members include functions, variables, and more.

In this example, we have two functions—Get-OSInformation and Get-DiskInformation. Let's save this as `Sysinformation.psm1` in the temp folder for now.

```

Function Get-SystemInformation
{
    param(
        [Parameter(Mandatory = $true)]
        [String]
        $ComputerName
    )
}
```

```

)
$OS = Get-WmiObject -Class Win32_OperatingSystem
$BIOS = Get-WmiObject -Class Win32_Bios
$CS = Get-WmiObject -Class Win32_ComputerSystem
$Properties = New-Object psobject -Property @{

    "OSName" = $OS.Caption
    "ServicePack" = $OS.CSDVersion
    "SerialNumber" = $BIOS.SerialNumber
    "Manufacturer" = $BIOS.Manufacturer
    "BootUpState" = $CS.BootupState
}
$Properties
}

Get-SystemInformation -ComputerName localhost

```

Now, we need to import a module; to do this we need to use the `Import-Module` command. In the following code the `-Verbose` and `-Force` parameters are used for a clear output. Take a look at the following image:

PS C:\> Import-Module C:\Temp\SysInformation.PSM1 -Verbose -Force
 VERBOSE: Removing the imported "Get-OSInformation" function. 5
 VERBOSE: Removing the imported "Get-DiskInformation" function.
 VERBOSE: Loading module from path 'C:\Temp\SysInformation.PSM1'.
 VERBOSE: Importing function 'Get-DiskInformation'. 6
 VERBOSE: Importing function 'Get-OSInformation'. 6

PS C:\> |

The points marked in the figure are explained in the following list:

- 1: Here, we use the `Import-Module` cmdlet to import the script module we created. The module name is `SysInformation.PSM1`.
- 2: Here, we explicitly mention the path—we have saved the script in `C:\Temp\`.
- 3: `-Verbose` is used to see the verbose data.
- 4: `-Force` parameter is used to remove the existing script and import it newly.
- 5: Since we use the force parameter we can see here that the

function is being removed.

- 6: Here, the function is being imported again.

Now, let's call the functions as cmdlet. Take a look at the following image:

The screenshot shows two PowerShell sessions. The top session runs `Get-DiskInformation -ComputerName .` (marked with a red circle 1) and displays disk space details for drives L:, R:, C:, D:, and S:. The bottom session runs `Get-OSInformation -ComputerName .` (marked with a red circle 2) and displays operating system information including architecture, serial number, and last boot time. Red callout boxes point from the numbers to their respective command outputs.

```
PS C:\> Get-DiskInformation -ComputerName .
+-----+
| Label FreeSpace(GB) Size(GB) |
+-----+
| L:    28.43      146.49   |
| R:    16.75      20.12    |
| C:    88.85      297.60   |
| D:    0.00       0.00     |
| L:    28.43      146.49   |
| R:    16.75      20.12    |
+-----+



PS C:\> Get-OSInformation -ComputerName .
+-----+
| Caption          OSArchitecture SerialNumber           LastBootUptime |
+-----+
| Microsoft Windows 7 Enterprise | 64-bit          00392-918-5000002-85838 | 6/17/2015 10:31:43 AM |
+-----+
```

Note

Note that each `Get-OSInformation` and `Get-DiskInformation` cmdlet has its own help information.

Exploring Windows PowerShell

5.0

Technet24.ir

In this section, we will cover the following topics:

- Basics of Desired State Configuration
- Parsing structured objects using PowerShell
- Exploring Package Management
- Exploring PowerShellGet
- Exploring other enhanced features

Windows PowerShell 5.0 has many significant benefits; to know more about them, refer to the following link:

<http://go.microsoft.com/fwlink/?LinkID=512808>

Here are a few highlights of Windows PowerShell 5.0:

- Improved usability
- Backward compatibility
- `Class` and `enum` keywords are introduced
- Parsing structured objects is easy using the `ConvertFrom-String` command
- We have a few new modules introduced in Windows PowerShell 5.0, such as `Archive`, `Package Management` (formerly known as `OneGet`), and so on
- ISE supports transcriptions
- Using `PowerShellGet`, we can find, install, and publish modules
- Debugging at runspace is possible using the `Microsoft.PowerShell.Utility` module

The basics of Desired State Configuration

Desired State Configuration, also known as DSC, is a new management platform in Windows PowerShell. Using DSC, we can deploy and manage the configuration data for software servicing and can also

manage the environment. DSC can be used to streamline a datacenter. This was introduced along with Windows Management Framework 4.0 and was heavily extended in Windows Management Framework 5.0.

Here are a few highlights of DSC in the April 2015 Preview:

- New cmdlets are introduced in WMF 5.0
- A few of the DSC commands are updated, which has made remarkable changes to the configuration management platform in PowerShell 5.0
- The DSC resources can be built using class, so there is no need for a MOF file

It's not mandatory to know PowerShell or to learn DSC, but it adds a great advantage.

This is similar to function, as we can use a configuration keyword; however, there is a huge difference because in DSC everything is declarative. This is a cool feature of Desired State Configuration. So, before beginning this exercise, I created a DSCDemo lab machine in the Azure cloud with Windows Server 2012, and it's available out of the box. So, the default PowerShell version we will use is 4.0.

For now, let's create and define a simple configuration that creates a file in the localhost. Yeah! A simple `New-Item` command can do this, but it's an imperative cmdlet, and we need to write a program to tell the computer to create it if it does not exist.

Let's execute the following code:

Structure of DSC configuration is as follows:

```
Configuration Name
{
    Node ComputerName
    {
        ResourceName <String>
        {
        }
    }
}
```

```
    }  
}
```

So now to create a simple text file with contents, we use the following code:

```
Configuration FileDemo  
{  
    Node $env:COMPUTERNAME  
    {  
        File FileDemo  
        {  
            Ensure = 'Present'  
            DestinationPath = 'C:\Temp\Demo.txt'  
            Contents = 'PowerShell DSC Rocks!'  
            Force = $true  
        }  
    }  
}
```

Take a look at the following image:

The screenshot shows a PowerShell script editor window titled "Untitled1.ps1*". The script contains the following code:

```
1 Configuration FileDemo ①  
2 { ②  
3     Node $env:COMPUTERNAME ③  
4     {  
5         File FileDemo ④  
6         {  
7             Ensure = 'Present'  
8             DestinationPath = 'C:\Temp\Demo.txt'  
9             Contents = 'PowerShell DSC Rocks!'  
10            Force = $true  
11        }  
12    }  
13 }  
14 FileDemo ⑤
```

Red circles with numbers 1 through 6 are placed over specific parts of the code to highlight them:

- 1: The first line of the Configuration block.
- 2: The opening brace of the Configuration block.
- 3: The first line of the Node block.
- 4: The first line of the File block.
- 5: The opening brace of the outer File block.
- 6: The closing brace of the outer File block.

The points marked in the figure are explained in the following list:

- 1: Here, using the `Configuration` keyword, we are defining a configuration with a name `FileDemo`—it's a friendly name
- 2: Inside the `Configuration` block, we created a `Node` block and created a file on localhost
- 3: Here, `File` is the resource name

- 4: Here, `FileDemo` is a friendly name of the resource, which is a string.
- 5: These are the properties of the file resource
- 6: This creates a MOF File—we called this similar to function; but wait, a file is not yet created here. We just created a MOF file.

Look at the MOF file structure in the following image:

```
/*
@TargetNode='DSCDEMOLAB'
@GeneratedBy=ChenV
@GenerationDate=06/23/2015 12:09:02
@GenerationHost=DSCDEMOLAB
*/

instance of MSFT_FileDirectoryConfiguration as $MSFT_FileDirectoryConfiguration1ref ②
{
    ResourceID = "[File]FileDemo";
    Ensure = "Present";
    Contents = "PowerShell DSC Rocks!";
    DestinationPath = "C:\\\\Temp\\\\Demo.txt";
    Force = True;
    ModuleName = "PSDesiredStateConfiguration";
    SourceInfo = "::5::9::File";
    ModuleVersion = "1.0";
};

instance of OMI_ConfigurationDocument ③
{
    Version="1.0.0";
    Author="ChenV";
    GenerationDate="06/23/2015 12:09:02";
    GenerationHost="DSCDEMOLAB";
};
```

We can manually edit the MOF file and use it on other machines that have PS 4.0 installed. It's not mandatory to use PowerShell to generate a MOF, but if you are comfortable with PowerShell, you can directly write the MOF file.

To explore the available DSC Resources, you can execute the following command:

`Get-DscResource`

The output is illustrated in the following image:

ImplementedAs	Name	Module	Properties
Binary	File	PSDesiredStateConfiguration	{DestinationPath, Attributes, Checksum, Content...}
Composite	1 FileDemo 2	PSDesiredStateConfiguration	{DependsOn}
PowerShell	Archive	PSDesiredStateConfiguration	{Destination, Path, Checksum, Credential...}
PowerShell	Environment	PSDesiredStateConfiguration	{Name, DependsOn, Ensure, Path...}
PowerShell	Group	PSDesiredStateConfiguration	{GroupName, Credential, DependsOn, Description...}
Binary	Log	PSDesiredStateConfiguration	{Message, DependsOn}
PowerShell	Package	PSDesiredStateConfiguration	{Name, Path, ProductId, Arguments...}
PowerShell	Registry	PSDesiredStateConfiguration	{Key, ValueName, DependsOn, Ensure...}
PowerShell	Script	PSDesiredStateConfiguration	{GetScript, SetScript, TestScript, Credential...}
PowerShell	Service	PSDesiredStateConfiguration	{Name, BuiltInAccount, Credential, DependsOn...}
PowerShell	User	PSDesiredStateConfiguration	{UserName, DependsOn, Description, Disabled...}
PowerShell	WindowsFeature	PSDesiredStateConfiguration	{Name, Credential, DependsOn, Ensure...}
PowerShell	WindowsProcess	PSDesiredStateConfiguration	{Arguments, Path, Credential, DependsOn...}

The points marked in the figure are explained in the following list:

- 1: This shows how the resources are implemented, which can be Binary, Composite, PowerShell, and so on. In the preceding example, we created a DSC configuration, that is `FileDemo` and that is listed as Composite.
- 2: This is the name of the resource.
- 3: This is the module name the resource belongs to.
- 4: These are the properties of resources.

To know the syntax of a particular DSC resource we can try the following code:

```
Get-DscResource -Name Service -Syntax
```

The output is illustrated in the following image which shows the resource syntax in detail:

```
PS C:\> Get-DscResource -Name Service -Syntax
Service [string] #ResourceName
{
    Name = [string]
    [ BuiltInAccount = [string] { LocalService | LocalSystem | NetworkService } ]
    [ Credential = [PSCredential] ]
    [ DependsOn = [string[]] ]
    [ StartupType = [string] { Automatic | Disabled | Manual } ]
    [ State = [string] { Running | Stopped } ]
}
```

Now, let's take a look at how DSC works and its three different phases, which are as follows:

- The Authoring phase
- The Staging phase
- The "Make it so" phase

The Authoring phase

In this phase, we will create a DSC configuration using PowerShell, and this will output a MOF file. We have seen a `FileDemo` example for creating a configuration. This is considered to be the Authoring phase.

The Staging phase

In this phase, the declarative MOF file will be staged as per its node (a MOF file will be created for each computer). DSC has the push and pull model, where push simply pushes configuration to the target nodes. The custom providers need to be manually placed in the target machines. On the other hand, in the pull model, we need to build an IIS server that will have the MOF files as the target nodes. This is well defined by the OData interface. In the pull model, the custom providers are downloaded to target system.

The "Make it so" phase

This is the phase used to enact the configuration. In other words, here we will apply the configuration on the target nodes.

We will cover the multinode configuration and the other DSC topic details in [Chapter 3, Exploring Desired State Configuration](#).

Before we summarize the basics of DSC, let's take a look at a few more DSC commands. We can do this by executing the following command:

```
Get-Command -Noun DSC*
```

The output is as shown in the following image:

CommandType	Name	ModuleName
Function	Get-DscConfiguration	PSDesiredStateConfiguration
Function	Get-DscLocalConfigurationManager	PSDesiredStateConfiguration
Function	Get-DscResource	PSDesiredStateConfiguration
Function	New-DscCheckSum	PSDesiredStateConfiguration
Function	Remove-DscConfigurationDocument	PSDesiredStateConfiguration
Function	Restore-DscConfiguration	PSDesiredStateConfiguration
Function	Stop-DscConfiguration	PSDesiredStateConfiguration
Function	Test-DscConfiguration	PSDesiredStateConfiguration
Cmdlet	Set-DscLocalConfigurationManager	PSDesiredStateConfiguration
Cmdlet	Start-DscConfiguration	PSDesiredStateConfiguration
Cmdlet	Update-DscConfiguration	PSDesiredStateConfiguration

Note

Note that we are using the PowerShell 4.0 stable release and not 5.0; so, the version property for the cmdlet will not be listed.

Local Configuration Manager (LCM) is the engine for DSC, and this runs on all the nodes. LCM is responsible for calling the configuration resources that are included in a DSC configuration script. Try executing the `Get-DscLocalConfigurationManager` cmdlet to explore its properties. To apply the LCM settings on the target nodes, we can use the `Set-DscLocalConfigurationManager` cmdlet.

Use case of classes in WMF 5.0

Using classes in PowerShell gets IT professionals, system administrators, and system engineers to start learning development.

Okay, it's time for us to switch back to Windows PowerShell 5.0 because the `Class` keyword is supported in the versions from 5.0 onward. You may wonder, why do we need to write class in PowerShell? Is it specially needed? Perhaps in this section, we will answer this, but this is one reason for which I would say that PowerShell is far better than scripting language.

When the class keyword was introduced, it was mainly focused at creating DSC resources. However, using class, we can create objects as we would with any other object-oriented programming language. The

class we will create in Windows PowerShell is truly a .NET framework type.

You may now wonder, how do we create a PowerShell class? It's easy: just use the `Class` keyword! The following steps will help you to create a PowerShell class:

1. Create a class using the command `Class ClassName { }`—This is an empty class.
2. Define properties in the class using the command `Class ClassName { $Prop1 , $prop2 }`
3. Instantiate the class using the command `$var = [ClassName]::New()`
4. Now, take a look at the output of `$var`:

```
Class ClassName {
    $Prop1
    $Prop2
}
$var = [ClassName]::new()
$var
```

In this example, we will take a look at how to create a class and what its advantages are.

Define Properties in Class; run the following command:

```
Class Catalog
{
    #Properties
    $Model = 'Fujitsu'
    $Manufacturer = 'Life Book S Series'
}

$var = New-Object Catalog
$var
```

The following image shows the output of class, its members, and setting of the property value:

```
PS C:\> Class Catalog
{
    #Properties
    $Model = 'Fujitsu'
    $Manufacturer = 'Life Book S Series'
}

$var = New-Object Catalog
$var
```

Model	Manufacturer
Fujitsu	Life Book S Series

Returns the properties

```
PS C:\> $var | GM
```

Results of Get Member

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
Manufacturer	Property	System.Object Manufacturer {get;set;}
Model	Property	System.Object Model {get;set;}

If we change the property value, the output will be as shown in the following image:

```
PS C:\> $var.Manufacturer = "Microsoft"
PS C:\> $var
```

Model	Manufacturer
Fujitsu	Microsoft

Allows to set the properties

Now, let's create a method with the overloads. In the following example, we will create a method named `SetInformation` that will accept two arguments, `$mdl` and `$mfgr`, and these are of the `String` type.

Using `$var.SetInformation`, and with no parentheses, we will take a

look at the overload definitions of the method.

Execute the following code:

```
Class Catalog
{
    #Properties
    $Model = 'Fujitsu'
    $Manufacturer = 'Life Book S Series'
    SetInformation([String]$mdl,[String]$mfgr)
    {
        $this.Manufacturer = $mfgr
        $this.Model = $mdl
    }
}
$var = New-Object -TypeName Catalog
$var.SetInformation
#Output
OverloadDefinitions
-----
void SetInformation(string mdl, string mfgr)
```

Let's set Model and Manufacturer using Set information, as follows:

```
Class Catalog
{
    #Properties
    $Model = 'Fujitsu'
    $Manufacturer = 'Life Book S Series'
    SetInformation([String]$mdl,[String]$mfgr)
    {
        $this.Manufacturer = $mfgr
        $this.Model = $mdl
    }
}
$var = New-Object -TypeName Catalog
$var.SetInformation('Surface' , 'Microsoft')
$var
```

The output is illustrated in the following image:

```

PS C:\> Class Catalog
{
    #Properties
    $Model = 'Fujitsu'
    $Manufacturer = 'Life Book S Series'

    SetInformation([String]$mdl,[String]$mfgr)
    {
        $this.Manufacturer = $mfgr
        $this.Model = $mdl
    }
}
$var = New-Object -TypeName Catalog
$var.SetInformation('Surface' , 'Microsoft') ←
$var

```

Result

Model	Manufacturer
Surface	Microsoft

In PowerShell class, we can use the PowerShell cmdlets as well. The following code is just a demo of using the PowerShell cmdlet.

Class allows us to validate the parameters as well. Let's take a look at the code in the following example:

```

Class Order
{
    [ValidateSet("Red" , "Blue" , "Green")]
    $color
    [ValidateSet("Audi")]
    $Manufacturer
    Book($Manufacturer , $color)
    {
        $this.color = $color
        $this.Manufacturer = $Manufacturer
    }
}

```

The \$Color and \$Manufacturer parameters have a ValidateSet attribute, which has a set of values. Now, let's use New-Object and set property with an argument that doesn't belong to this set. Run the following command:

```
$var = New-Object Order  
$var.color = 'Orange'
```

Now, we will get the following error:

```
Exception setting "color": "The argument "Orange" does not  
belong to the set "Red,Blue,Green" specified by the ValidateSet  
attribute. Supply an argument that is in the set and then try  
the command again."
```

Let's set the argument values correctly to get the result using the Book method:

```
$var = New-Object Order  
$var.Book('Audi' , 'Red')  
$var
```

The output is illustrated in the following image:

```
$var = New-Object Order  
$var.Book('Audi' , 'Red')  
$var  
  
color Manufacturer  
-----  
Red Audi
```

Constructors

A constructor is a special type of method that creates new objects. It has the same name as the class, and the return type is `void`. Multiple constructors are supported, but each one has to take different numbers and types of parameters. In this exercise, let's take a look at the steps to create a simple constructor in PowerShell that will create a user in the active directory. Execute the following code:

```
Class ADUser {  
    $identity  
    $Name  
    ADUser($Identity , $Name) {  
        New-ADUser -SamAccountName $Identity -Name $Name
```

```

    $this.identity = $Identity
    $this.Name = $Name
}
}

$var = [ADUser]::new('Dummy' , 'Test Case User')
$var

```

The output of the code we just saw is illustrated in the following image:

A screenshot of a PowerShell window. The command run is:

```
PS C:\> $var = [ADUser]::new('Dummy' , 'Test Case User')
```

The output shows a table with two rows:

identity	Name
Dummy	Test Case User

A red speech bubble points to the second row with the text "Created a user in AD".

We can also hide the properties in a PowerShell class; as an example, let's create two properties and hide one. In theory, it hides the property but we can still use it. Execute the following code:

```

Class Hide {
    [String]$Name
    Hidden $ID
}
$var = [Hide]::new()
$var

```

A screenshot of a PowerShell window. The command run is:

```
PS C:\> $var = [Hide]::new()
```

The output shows a table with one row:

identity	Name
	Dummy

A red speech bubble points to the word "Hidden" in the class definition with the text "Hidden".

Below the table, the variable \$var is expanded to show its properties:

- Name (highlighted in blue)
- Equals
- GetHashCode
- GetType
- ToString

A red speech bubble points to the "Name" property with the text "Not shown here!".

However, we can perform operations such as `Get` and `Set`, as shown in the following code:

```
Class Hide {
    [String]$Name
    Hidden $ID
}
$var = [Hide]::new()
$var.Id = '23'
$var.Id
```

This will return the output as 23.

To know more about the class, use `help about_Classes -Detailed`.

Parsing structured objects using PowerShell

In Windows PowerShell 5.0, a new cmdlet, `ConvertFrom-String`, has been introduced, and this is available in the `Microsoft.PowerShell.Utility` module.

Using this command, we can parse structured objects from any given string content. For more information, use `help ConvertFrom-String -Detailed`.

Note

Note that the help has an incorrect parameter, `PropertyName`. Copying and pasting will not work, so you need to use the `ConvertFrom-String -Parameter * help` command and read the parameter—it's actually `PropertyNames`.

Now, let's consider an example where we will use `ConvertFrom-String`.

The scenario is such that a team has custom code that generates a log file to create a daily health checkup report of its environment. Unfortunately, the tool delivered by the vendor is an EXE file, and there is no source code available. The log file format is as follows:

```
"Error 4356 Lync" , "Warning 6781 SharePoint" , "Information 5436 Exchange",
"Error 3432 Lync" , "Warning 4356 SharePoint" , "Information 5432 Exchange"
```

There are many ways to manipulate this record, but let's take a look at how the PowerShell cmdlet, `ConvertFrom-String`, can help us. Using the following code, we will simply extract Type, EventID, and Server:

```
"Error 4356 Lync" , "Warning 6781 SharePoint" , "Information  
5436 Exchange",  
"Error 3432 Lync" , "Warning 4356 SharePoint" , "Information  
5432 Exchange" |  
ConvertFrom-String -PropertyNames Type , EventID, Server
```

So what's interesting in this? It's cool because now your output is a `PSCustomObject` object, which you can manipulate as required. Take a look at the following image:

PS C:\> "Error 4356 Lync" , "Warning 6781 SharePoint" , "Information 5436 Exchange",
"Error 3432 Lync" , "Warning 4356 SharePoint" , "Information 5432 Exchange" |
ConvertFrom-String -PropertyNames Type , EventID, Server | GM

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
EventID	NoteProperty	int16 EventID=4356
Server	NoteProperty	string Server=Lync
Type	NoteProperty	string Type=Error

Annotations:

- Annotation 1: A red circle with the number 1 is placed over the command `GM`.
- Annotation 2: A red circle with the number 2 is placed over the `Type` column header.
- Annotation 3: A red circle with the number 3 is placed over the `Type` value for the `EventID` row.

Following is the output we just saw in the image:

```
"Error 4356 Lync" , "Warning 6781 SharePoint" , "Information  
5436 Exchange",  
"Error 3432 SharePoint" , "Warning 4356 SharePoint" ,  
"Information 5432 Exchange" |  
ConvertFrom-String -PropertyNames Type , EventID, Server | ?  
{$_._Type -eq 'Error'}
```

One of the outputs is `Lync`, and SharePoint has some error logs that need to be looked at as a priority. As your requirements may vary, you can use this cmdlet in whichever way you need.

`ConvertFrom-String` has a delimiter parameter that helps us to manipulate the strings as well. In the following example, let's use the `-Delimiter` parameter, which removes white spaces and returns the properties:

```
"Chen V" | ConvertFrom-String -Delimiter "\s" -PropertyNames "FirstName" , "SurName"
```

This outputs `FirstName` and `SurName` as the result—`FirstName` as Chen and `SurName` as v.

In this example, we will walk you through using the template file to manipulate the string as we need. To do this, we need to use the `TemplateContent` parameter. You can use `help ConvertFrom-String -Parameter TemplateContent`.

Before we begin, we need to create a template file; to do this, let's ping a website, `www.microsoft.com`, which will return the outputs as follows:

```
Pinging e10088.dspb.akamaiedge.net [2.21.47.138] with 32 bytes of data:  
Reply from 2.21.47.138: bytes=32 time=37ms TTL=51  
Reply from 2.21.47.138: bytes=32 time=35ms TTL=51  
Reply from 2.21.47.138: bytes=32 time=35ms TTL=51  
Reply from 2.21.47.138: bytes=32 time=36ms TTL=51  
Ping statistics for 2.21.47.138:  
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),  
Approximate round trip times in milli-seconds:  
    Minimum = 35ms, Maximum = 37ms, Average = 35ms
```

Now that we have the information in some structure, let's extract the IP and the bytes to do this. I replaced the IP and the bytes with `{IP*:2.21.47.138}`, as follows:

```
Pinging e10088.dspb.akamaiedge.net [2.21.47.138] with 32 bytes of data:  
Reply from {IP*:2.21.47.138}: bytes={[int32]Bytes:32} time=37ms TTL=51  
Reply from {IP*:2.21.47.138}: bytes={[int32]Bytes:32} time=35ms TTL=51  
Reply from {IP*:2.21.47.138}: bytes={[int32]Bytes:32} time=36ms TTL=51
```

```

Reply from {IP*:2.21.47.138}: bytes={[int32]Bytes:32} time=35ms
TTL=51
Ping statistics for 2.21.47.138:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 35ms, Maximum = 37ms, Average = 35ms

```

The output is illustrated in the following figure:

```

PS C:\> ping www.microsoft.com | ConvertFrom-String -TemplateFile C:\Temp\Template.txt

IP          Bytes
--          -----
2.21.47.138 32
2.21.47.138 32
2.21.47.138 32
2.21.47.138 32

```

`ConvertFrom-String` has a `Debug` parameter. Using this, we can debug our template file. In the following example, let's consider the debugging output:

```

ping www.microsoft.com | ConvertFrom-String -TemplateFile
C:\Temp\Template.txt -Debug

```

```

PS C:\> ping www.microsoft.com | ConvertFrom-String -TemplateFile C:\Temp\Template.txt -Debug
DEBUG: Property: IP
Program: <SequenceProgram symbol="NL" score="5481" ><NonterminalNode symbol="$NL" rule="LinesMap">
<NonterminalNode symbol="$T1" rule="SingleLinePositionPair">
<VariableNode symbol="$v" />
<NonterminalNode symbol="$P1" rule="RegPos1">
<VariableNode symbol="$s1" />
<NonterminalNode symbol="$RR" rule="RegexPair">
<LiteralNode symbol="$r">
<RegularExpression />
</LiteralNode>
<LiteralNode symbol="$r">
<RegularExpression>
<Token name="Date" score="109" isSymbol="true"><! [CDATA[((?<!\\d)(\\d?\\d)([-\\\\.])\\d?\\d)\\3(19|20)?\\d\\d(?\\d) | (?<!\\d)(19|20)?\\d\\d
([-\\\\.])\\d?\\d)\\/(\\d?\\d)(?!\\d)]]></Token>
</RegularExpression>
</LiteralNode>
<NonterminalNode>
<LiteralNode symbol="$k"><! [CDATA[1]]></LiteralNode>
</NonterminalNode>
<NonterminalNode symbol="$P1" rule="RegPos1">
<VariableNode symbol="$s1" />
<NonterminalNode symbol="$RR" rule="RegexPair">
<LiteralNode symbol="$r">
<RegularExpression>
<Token name="IP" score="109" isSymbol="true"><! [CDATA[(?<!\\d)(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?).){3}(?:25[0-5]|2[0-4]
[0-9]| [01]?[0-9][0-9]?)(?!\\d)]]></Token>

```

Note

Note that as we mentioned earlier, PowerShell 5.0 is a preview release and has a few bugs. Let's ignore these for now and focus on the features

that work fine and can be utilized in our environment.

Exploring Package Management

In this topic, we will walk you through the features of Package Management. This is another great feature of Windows Management Framework 5.0. This is introduced in Windows 10, and formerly, Package Management was known as OneGet.

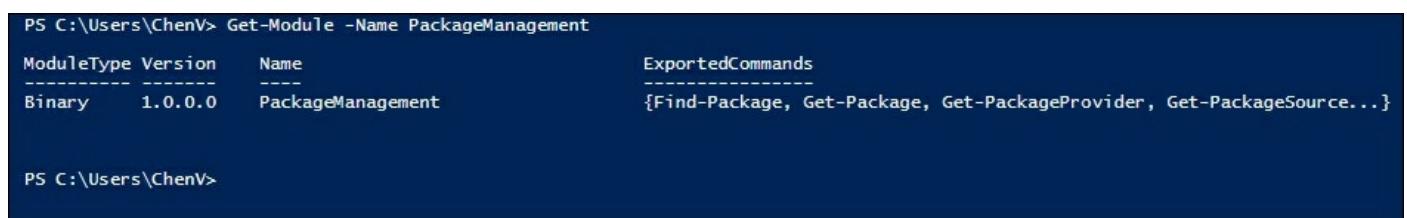
Using Package Management, we can automate software discovery, installation of software, and inventorying. Do not think about **Software Inventory Logging (SIL)** for now; we will cover that later in [Chapter 3, Exploring Desired State Configuration](#).

As we know, the Windows software installation technology has its own way of installing, for example MSI, MSU and so on. It is a real challenge for IT professionals and DevOps to think about unique automation of software installation / deployments. Now, we can do it using the Package Management module.

To begin with, let's take a look at the Package Management module using the following code:

```
Get-Module -Name PackageManagement
```

The output is illustrated in the following image:



ModuleType	Version	Name	ExportedCommands
Binary	1.0.0.0	PackageManagement	{Find-Package, Get-Package, Get-PackageProvider, Get-PackageSource...}

Well, we've got an output showing that it's a binary module. So, how do we find the available cmdlets and their usage? PowerShell always has the simplest way of doing things! Execute the following code:

```
Get-Module -Name PackageManagement
```

The available cmdlets are shown in the following image:

CommandType	Name	Version	Source
Cmdlet	Find-Package	1.0.0.0	PackageManagement
Cmdlet	Get-Package	1.0.0.0	PackageManagement
Cmdlet	Get-PackageProvider	1.0.0.0	PackageManagement
Cmdlet	Get-PackageSource	1.0.0.0	PackageManagement
Cmdlet	Install-Package	1.0.0.0	PackageManagement
Cmdlet	Register-PackageSource	1.0.0.0	PackageManagement
Cmdlet	Save-Package	1.0.0.0	PackageManagement
Cmdlet	Set-PackageSource	1.0.0.0	PackageManagement
Cmdlet	Uninstall-Package	1.0.0.0	PackageManagement
Cmdlet	Unregister-PackageSource	1.0.0.0	PackageManagement

Package providers are nothing but providers connected to Package Management (OneGet), and package sources are registered for the providers. To view the list of providers and sources, we will use the following cmdlets:

PS C:\Users\ChenV> Get-PackageProvider
Name

Programs
msu
msi
Chocolatey
NuGet
PSModule

PS C:\Users\ChenV> Get-PackageSource
Name

chocolatey
PSGallery

Now, let's take a look at the packages available. In the following example, I will select the first 20 packages just for an easy view:

PS C:\> Find-Package Select -First 20
Name

GallioBundle
scummvm
residualvm
WindowsAzurePowerShell_0871
Leechcraft
AzureBuildsDKvs2013
X.Sockets.PerformanceCounters
GoCiAgent
TinQPad4.AnyCPU.portable
poshpuppetreports
ceed
simply-weather
OrchardCms
frozenbytes.dev.essentials
csved
dtksnack
jpegoptim
resourcesextract
xbox-one-controller
tfsSidekicks

Version

3.4
1.7.0
0.1.1
0.8.7.1
0.6.70.2
2.4.0
0.1
14.2.0
4.51.03
0.9.55
0.8.0
0.2.2
1.8.1.2
1.01
2.3.2
1.0.0.1
1.2.2.4.4
1.18
1.0.0
2.4.0

Source

chocolatey

Summary

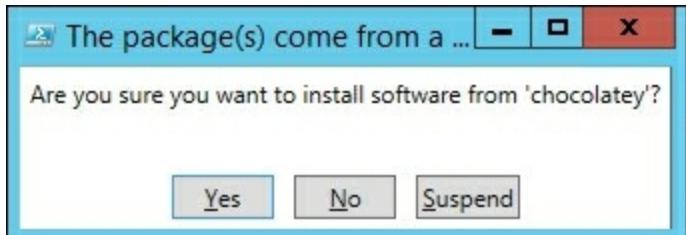
The Gallio platform seeks to facilitate the creation of a rich ecosystem of interoperable
scummvm
residualvm
LeechCraft is a free open source cross-platform modular live environment.
Azure SDK for build servers and Visual Studio 2013
Installs Performance Monitor counters for X.Sockets.NET v4
Installs Go Agent
LINQPad 4 (Portable, AnyCPU build for "massive" queries)
ceed
Weather without all the fuss
Installs an instance of the Orchard CMS
frozenbytes.dev.essentials
CSVed is an easy and powerful CSV file editor, you can manipulate any CSV file, separated
The DTF Barcode Reader SDK loves to pop up a licensing dialog when running in debug.
Utility to optimize jpeg files. Provides lossless optimization (based on optimizing the I
Extract resource files (bitmaps, icons, html files, and more) from dll files
PC Drivers for the Xbox One Controller Now Available
Team Foundation Sidekicks is a suite of tools for Microsoft Team Foundation Server admin

Now, we have 20 packages; using the `Install-Package` cmdlet, let's install `WindowsAzurePowerShell` on our Windows 2012 Server.

We need to ensure that the source is available prior to any installations. To do that, simply execute the `Get-PackageSource` cmdlet. If the source, **chocolatey**, doesn't show up in the output, simply execute the following code; however, do not change any values. This code will install chocolatey in your machine. Once the installation is done, we need to restart PowerShell:

```
Invoke-Expression ((New-Object  
System.Net.WebClient).DownloadString('https://chocolatey.org/in  
stall.ps1'))  
Find-Package -Name WindowsAzurePowerShell | Install-Package -  
Verbose
```

The preceding command shows the confirmation dialog for chocolatey, which is the package source, as shown in the following screenshot:



Click on **Yes** to install the package.

```
PS C:\> Find-Package -Name WindowsAzurePowerShell | Install-Package -Verbose  
VERBOSE: Performing the operation "Install Package" on target "Package 'WindowsAzurePowershell' version '0.8.8' from 'chocolatey'".  
VERBOSE: NuGet: GET http://chocolatey.org/api/v2/Packages(Id='DotNet4.5',Version='4.5')  
VERBOSE: NuGet: GET http://chocolatey.org/api/v2/package/DotNet4.5/4.5  
VERBOSE: NuGet: Installing 'DotNet4.5 4.5'.  
VERBOSE: NuGet: Successfully installed 'DotNet4.5 4.5'.  
VERBOSE:  
VERBOSE: False 1  
VERBOSE: NuGet: GET http://chocolatey.org/api/v2/Packages(Id='WindowsAzurePowershell',Version='0.8.8')  
VERBOSE: NuGet: Attempting to resolve dependency 'DotNet4.5'.  
VERBOSE: NuGet: GET http://chocolatey.org/api/v2/package/WindowsAzurePowershell/0.8.8  
VERBOSE: NuGet: Installing 'WindowsAzurePowershell 0.8.8'.  
VERBOSE: NuGet: Successfully installed 'WindowsAzurePowershell 0.8.8'.  
VERBOSE: CreateFolder Success C:\Users\Chen\AppData\Local\Temp\2\chocolatey\WindowsAzurePowershell 2  
VERBOSE: GetChocolateyWebFile WindowsAzurePowershell => http://az412849.vo.msecnd.net/downloads03/azure-powershell1.0.8.8.msi  
VERBOSE: Package Successfully Installed WindowsAzurePowershell  
VERBOSE: True 3  
Name Version Source Summary  
---- ----   
DotNet4.5 4.5 chocolatey The Microsoft .NET Framework 4.5  
WindowsAzurePowerShell 0.8.8 chocolatey
```

Installed .NET 4.5 and Windows Azure PowerShell

The points marked in the figure are explained in the following list:

- 1: Here, the prerequisites are installed
- 2: Here, a temp folder is created
- 3: Here, the package is installed successfully

Windows Server 2012 has .NET 4.5 in the box by default, so the verbose will turn up as `False` for .NET 4.5. This means that .NET 4.5 will be skipped while enacting the configuration, but `WindowsAzurePowerShell` will be installed successfully. Take a look at the following image:



If you try to install the same package and the same version that is available in your system, the cmdlet will skip the installation. Execute the following code:

```
Find-Package -Name PowerShellHere | Install-Package -Verbose  
VERBOSE: Skipping installed package PowerShellHere 0.0.3
```

Explore all the Package Management cmdlets and automate your software deployments.

Exploring PowerShellGet

PowerShellGet is a module available in the Windows PowerShell 5.0 preview. You can do the following using this:

- Search through the modules in the Gallery with `Find-Module`
- Save modules to your system from the Gallery with `Save-Module`
- Install modules from the Gallery with `Install-Module`
- Update your modules to the latest version with `Update-Module`
- Add your own custom repository with `Register-PSRepository`

The following screenshot shows the additional cmdlets that are available:

```
PS C:\> Import-Module 'C:\Program Files\WindowsPowerShell\Modules\PowerShellGet' -Verbose
VERBOSE: Loading module from path 'C:\Program Files\WindowsPowerShell\Modules\PowerShellGet\PowerShellGet.psd1'.
VERBOSE: Loading 'FormatsToProcess' from path 'C:\Program Files\WindowsPowerShell\Modules\PowerShellGet\PSGet.Format.ps1xml'.
VERBOSE: Loading module from path 'C:\Program Files\WindowsPowerShell\Modules\PowerShellGet\PSGet.psm1'.
VERBOSE: Importing function 'Find-Module'.
VERBOSE: Importing function 'Get-InstalledModule'.
VERBOSE: Importing function 'Get-PSRepository'.
VERBOSE: Importing function 'Install-Module'.
VERBOSE: Importing function 'Publish-Module'.
VERBOSE: Importing function 'Register-PSRepository'.
VERBOSE: Importing function 'Save-Module'.
VERBOSE: Importing function 'Set-PSRepository'.
VERBOSE: Importing function 'Uninstall-Module'.
VERBOSE: Importing function 'Unregister-PSRepository'.
VERBOSE: Importing function 'Update-Module'.
VERBOSE: Importing alias 'fimo'.
VERBOSE: Importing alias 'inmo'.
VERBOSE: Importing alias 'pumo'.
VERBOSE: Importing alias 'upmo'.
```

You can refer to this link for more information about PowerShell Gallery:

<https://www.powershellgallery.com>

This will allow us to find a module from PowerShell Gallery and install it in our environment. PSGallery is a repository of modules. Using the `Find-Module` cmdlet, we can obtain a list of the modules available in PS Gallery. Pipe and install the required module. Alternatively, we can save the module and examine it before installation; to do this, you need to use the `Save-Module` cmdlet.

The following image illustrates the installation and uninstallation of the

xJEA module:

```
PS C:\> Find-Module -Name xJEA | Install-Module -Verbose
VERBOSE: The specified Location is 'https://www.powershellgallery.com/api/v2/' and PackageManagementProvider is 'NuGet'.
VERBOSE: Getting the provider object for the PackageManagement Provider 'NuGet'.
VERBOSE: The specified Location is 'https://www.powershellgallery.com/api/v2/' and PackageManagementProvider is 'NuGet'.
VERBOSE: Performing the operation "Install-Module" on target "Version '0.2.16.6' of module 'xJea'".
VERBOSE: The specified module will be installed in 'C:\Program Files\WindowsPowerShell\Modules'.
VERBOSE: The specified Location is 'NuGet' and PackageManagementProvider is 'NuGet'.
VERBOSE: Downloading module 'xJea' with version '0.2.16.6' from the repository 'https://www.powershellgallery.com/api/v2/'.
VERBOSE: NuGet: GET https://www.powershellgallery.com/api/v2/Packages(Id='xJea',Version='0.2.16.6')
VERBOSE: NuGet: GET https://www.powershellgallery.com/api/v2/package/xJea/0.2.16.6
VERBOSE: NuGet: Installing 'xJea 0.2.16.6'.
VERBOSE: NuGet: Successfully installed 'xJea 0.2.16.6'.
VERBOSE: Module 'xJea' was installed successfully.

PS C:\> Uninstall-Module -Name xJEA -Verbose
VERBOSE: Performing the operation "Uninstall-Module" on target "Version '0.2.16.6' of module 'xJea'".
VERBOSE: Successfully uninstalled the module 'xJea' from module base 'C:\Program Files\WindowsPowerShell\Modules\xJea\0.2.16.6'.

PS C:\> |
```

We can also publish a module in PSGallery, which will be available over the internet for others.

This is not a great module; all it does is get user information from active directory for the same given account name, so you can create a function and save it as PSM1 in the module folder. In order to publish the module in PSGallery, we need to ensure that the module manifests. We will take a look at the details of the modules in the next chapter.

Perform the following steps:

1. Create a PSM1 file.
2. Create a PSD1 file which is a Manifest Module—data file.
3. Get your NuGet API key from the PSGallery link shared before.
4. Publish your module using the `Publish-Module` cmdlet.

We get the following output:

Manage My Modules

Modules

These modules are currently published for the world to see.

Module	Module ID	Description	Downloads
  Testing	Testing	Just for Demo!	0
You have a total of 1 modules.			0

Following figure shows the published module:

```
PS C:\> Publish-Module -Name Testing -NuGetApiKey FF9b122F-067F-4a45-975F-55df02855a6e -Verbose
VERBOSE: Repository details, Name = ' PSGallery ', Location = ' https://www.powershellgallery.com/api/v2/' ; IsTrusted = 'True' ; IsRegistered = 'True'.
VERBOSE: Repository details, Name = ' PSGallery ', Location = ' https://www.powershellgallery.com/api/v2/' ; IsTrusted = 'True' ; IsRegistered = 'True'.
VERBOSE: Module 'Testing' was found in 'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Testing'.
VERBOSE: Repository details, Name = ' PSGallery ', Location = ' https://www.powershellgallery.com/api/v2/' ; IsTrusted = 'True' ; IsRegistered = 'True'.
VERBOSE: Repository details, Name = ' PSGallery ', Location = ' https://www.powershellgallery.com/api/v2/' ; IsTrusted = 'True' ; IsRegistered = 'True'.
VERBOSE: Using the specified source names : ' PSGallery '.
VERBOSE: Getting the provider object for the PackageManagement Provider ' NuGet '.
VERBOSE: The specified location is ' https://www.powershellgallery.com/api/v2/' and PackageManagementProvider is ' NuGet '.
VERBOSE: Performing the operation " Publish-Module " on target " Version '1.0' of module ' Testing ' ".
VERBOSE: Successfully published module ' Testing ' to the module publish location ' https://go.microsoft.com/fwlink/?LinkID=397527&clcid=0x409 ' . Please allow few minutes for ' Testing ' to show up in the search results.
PS C:\>
```

Understanding PowerShell modules

Windows PowerShell modules present a simple way to organize and package our script so as to make it distributable and reusable.

In this section, we will cover the following topics:

- What is a module?
- Script modules
- Binary modules
- Manifest modules
- Dynamic modules

Introduction to modules

Windows PowerShell can be created dynamically or persisted to our disk. While discussing PowerShell modules, for a moment I thought about the old-style snap-ins in PowerShell version 1.0, which still exist. Yeah! SharePoint still has snap-ins and not modules.

Snap-ins only contain cmdlets and providers, whereas in modules, we can have functions, variables, aliases, and PowerShell drives.

Before we create a module, we need to know what a module should deliver and where it should be placed. Use this code to find the module path:

```
$env:PSModulePath -split ','
```

You will see two different paths, where one is the system location and the other is the user profile location, as follows:

```
$ENV:Windir\System32\WindowsPowerShell\v1.0\Modules  
$ENV:UserProfile\Documents\WindowsPowerShell\Modules
```

The folder name and module name should be the same. For example, if

you create an `ADUserInformation` module, you need to save it under a folder named `ADUserInformation`.

Script modules

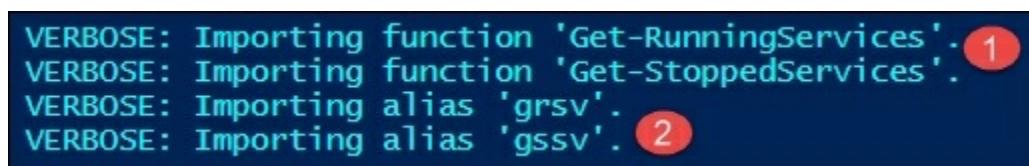
Any valid PowerShell code can be made as a script module. You can create a PowerShell script with a bunch of functions and save it as a PSM1 file, and that would be a script module.

Normally, we save the PowerShell script as PS1; if you save it as PSM1, you can't run the code as a script. Instead, we would need to use the `Import-Module` cmdlet, load the module to the disk, and make use of the cmdlets. The cmdlets here are nothing but your function name, as shown in the following code:

```
Function Get-RunningServices
{
    (Get-Service) .Where({$_.Status -eq 'Running'})
}

Function Get-StoppedServices
{
    (Get-Service) .Where({$_.Status -eq 'Stopped'})
}
New-Alias -Name grsv -Value Get-RunningServices
New-Alias -Name gssv -Value Get-StoppedServices
Export-ModuleMember -Function * -Alias *
```

Now, we will use the `Import-Module` command to load the module. Take a look at the following image:



The screenshot shows a PowerShell window with the following text output:

```
VERBOSE: Importing function 'Get-RunningServices'. 1
VERBOSE: Importing function 'Get-StoppedServices'.
VERBOSE: Importing alias 'grsv'.
VERBOSE: Importing alias 'gssv'. 2
```

Two points are highlighted with red circles: point 1 is over the first 'VERBOSE' line, and point 2 is over the last 'VERBOSE' line.

The points marked in the figure are explained in the following list:

- 1: Here, all the functions are imported
- 2: Here, all the aliases are imported

The following code will show you the aliases and functions that we will create in the PSM1 file:

```
Get-Module WindowsService | Select ExportedAliases ,  
ExportedFunctions
```

Using Windows PowerShell 5.0, we can write a class, save it as PSM1, and explore the objects. In the following example, we can take a look at how this works:

```
Class Demo  
{  
    $FirstName  
    $SurName  
    GetInformation($ID)  
    {  
        $result = Get-ADUser -Identity $ID  
        $this.FirstName = $result.GivenName  
        $this.SurName = $result.SurName  
    }  
}
```

The preceding code will result as shown in the following image:

A screenshot of a PowerShell window. The command `Import-Module C:\Temp\Class.psm1 -Verbose` is run, followed by `$var = New-Object Demo`. Then, `$var.GetInformation` is partially typed, and the output shows a table with columns `FirstName` and `SurName`, containing the values `Chendrayan` and `Venkatesan`. A callout bubble points to the parameter in the command with the text: "Pass SamaccountName as Parameter ('ID')".

```
PS C:\> Import-Module C:\Temp\Class.psm1 -Verbose  
VERBOSE: Loading module from path 'C:\Temp\Class.psm1'.  
PS C:\> $var = New-Object Demo  
PS C:\> $var.GetInformation  
PS C:\> $var  
FirstName SurName  
----- -----  
Chendrayan Venkatesan
```

Pass SamaccountName as Parameter ('ID')

Binary modules

A binary module is an assembly file (the DLL file). Inbuilt PowerShell modules are good examples of binary modules. Execute the following code:

```
Get-Module -Name Microsoft.PowerShell.*
```

Binary modules are faster and easier to build using a PowerShell reference. We can parameterize, validate the parameters, and perform many such functions in binary modules. A great feature in Visual Studio is that while parameterizing, we can get a syntax highlighter, which makes our job easier and faster.

The following exercise will guide us to creating a binary module. It's simple, and all this does is clear the temp files and the temp IE files. For this, we will create three cmdlets and compile them as DLL.

Perform the following steps to create a binary module:

1. Open Visual Studio.
2. Choose **Visual C#** and select **Class Library**.
3. Name your project.
4. Add `System.Management.Automation` DLL as a reference. This is available in your system's GAC assembly folder.
5. Add the code shared in the following section in `Class.cs`.
6. Create the solution, and now we will have the DLL file.
7. Import the DLL file and make use of the PowerShell cmdlet.

Let's take a look at the usage and explanation:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Management;
using System.Management.Automation;
using System.IO;

namespace Windows_Management
{
    [Cmdlet(VerbsCommon.Clear, "TemporaryInternetFiles")]
    public class WindowsManagement : PSCmdlet
    {
        protected override void ProcessRecord()
        {
            //Delete Internet Cache Files and Folders
```

```

        string path =
Environment.GetFolderPath(Environment.SpecialFolder.InternetCache);
        Console.ForegroundColor = ConsoleColor.DarkYellow;
        Console.WriteLine("Clearing Temporary Internet
Cache Files and Directories....." + path);
        System.IO.DirectoryInfo folder = new
 DirectoryInfo(path);
        foreach (FileInfo files in folder.GetFiles())
        {
            try
            {
                files.Delete();
            }
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine(ex);
            }
        }
        foreach (DirectoryInfo Directory in
folder.GetDirectories())
        {
            try
            {
                Directory.Delete();
            }
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine(ex);
            }
        }
        Console.WriteLine("Done Processing!!!!");
        Console.ResetColor();
    }
}
namespace clearInternetexplorerHistory
{
    [Cmdlet(VerbsCommon.Clear, "IEHistory")]
    public class clearInternetexplorerHistory : PSCmdlet
    {
        protected override void ProcessRecord()
        {
            // base.ProcessRecord();
            string path =
Environment.GetFolderPath(Environment.SpecialFolder.History);

```

```

Console.ForegroundColor = ConsoleColor.DarkYellow;
Console.WriteLine("Clearing Internet Explorer
History....." + path);
System.IO.DirectoryInfo folder = new
DirectoryInfo(path);
foreach (FileInfo files in folder.GetFiles())
{
    try
    {
        files.Delete();
    }
    catch (Exception ex)
    {
        System.Diagnostics.Debug.WriteLine(ex);
    }
}
foreach (DirectoryInfo Directory in
folder.GetDirectories())
{
    try
    {
        Directory.Delete();
    }
    catch (Exception ex)

    {
        System.Diagnostics.Debug.WriteLine(ex);
    }
}
Console.WriteLine("Done Processing!!!!");
Console.ResetColor();
}
}

namespace UserTemporaryFiles
{
[Cmdlet(VerbsCommon.Clear, "UserTemporaryFiles")]
public class UserTemporaryFiles : PSCmdlet
{
protected override void ProcessRecord()
{
    //base.ProcessRecord();
    string temppath = System.IO.Path.GetTempPath();
    System.IO.DirectoryInfo usertemp = new
    DirectoryInfo(temppath);
    Console.WriteLine("Clearing Your Profile Temporary

```

```

        Files..." + tempPath);
        foreach (FileInfo tempFiles in userTemp.GetFiles())
        {
            try
            {
                tempFiles.Delete();
            }
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine(ex);
            }
        }
        Console.WriteLine("Done Processing!!!");
        foreach (DirectoryInfo tempDirectory in
userTemp.GetDirectories())
        {
            try
            {
                tempDirectory.Delete();
            }
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine(ex);
            }
        }
    }
}

```

The screenshot shows a PowerShell window with the following command and output:

```

PS C:\> Import-Module C:\SystemManagement.dll -Verbose
VERBOSE: Loading module from path 'C:\SystemManagement.dll'. 1
VERBOSE: Importing cmdlet 'Clear-TemporaryInternetFiles'.
VERBOSE: Importing cmdlet 'Clear-IEHistory'.
VERBOSE: Importing cmdlet 'Clear-UserTemporaryFiles'.

```

Then, the following cmdlets are run:

- 3** PS C:\> Clear-TemporaryInternetFiles 3
Clearing Temporary Internet Cache Files and Directories.....C:\Users\ChenV\AppData\Local\Microsoft\Windows\INetCache
Done Processing!!!
- 4** PS C:\> Clear-IEHistory 4
Clearing Internet Explorer History.....C:\Users\ChenV\AppData\Local\Microsoft\Windows\History
Done Processing!!!
- 5** PS C:\> Clear-UserTemporaryFiles 5
Clearing Your Profile Temporary Files...C:\Users\ChenV\AppData\Local\Temp\2\
Done Processing!!!

- 1: Here, the DLL is loaded using the `Import-Module` cmdlet
- 2: Here, the cmdlet is loaded that we coded in the C# class library
- 3: The cmdlet `Clear-TemporaryInternetFiles` clears the temp IE

files

- 4: The cmdlet `Clear-IEHistory` clears the IE history
- 5: The cmdlet `Clear-UserTemporaryFiles` clears the user temporary files

Manifest modules

A manifest module is nothing but a data file with a PSD1 extension. This file will describe the contents of the module and how the module process works.

The manifest module is used to export an assembly that is installed in the global assembly cache. A manifest module is also required for modules that support the **Updatable Help** feature. Updatable Help uses the `HelpInfoUri` key in the manifest module to find the help information (HelpInfo XML) file, which contains the location of the updated help files for the module.

A manifest module can be created using PowerShell cmdlets. You can even manually type and save the file as PSD1, but using PowerShell is quicker. In the following example, we will create a manifest module:

```
$Param = @{
    Author = "Chendrayan Venkatesan"
    CompanyName = "Contoso"
    ModuleVersion = "1.0"
    Description = "Module Manifest Demo"
    PowerShellVersion = "5.0"
    Path = "C:\Temp\ModuleManifestDemo.psd1"
}
New-ModuleManifest @Param
```

The preceding code creates a manifest module, as shown in the following image:

```
#  
# Module manifest for module 'ModuleManifestDemo'  
#  
# Generated by: Chendrayan Venkatesan  
#  
# Generated on: 6/26/2015  
#  
#{@  
  
# Script module or binary module file associated with this manifest.  
# RootModule = ''  
  
# Version number of this module.  
ModuleVersion = '1.0'  
  
# ID used to uniquely identify this module  
GUID = 'abfe91d1-ef13-4d24-8b0f-26642f28eedf'  
  
# Author of this module  
Author = 'Chendrayan Venkatesan'  
  
# Company or vendor of this module  
CompanyName = 'Contoso'  
  
# Copyright statement for this module  
Copyright = '(c) 2015 Chendrayan Venkatesan. All rights reserved.'  
  
# Description of the functionality provided by this module  
Description = 'Module Manifest Demo'  
  
# Minimum version of the Windows PowerShell engine required by this module  
PowerShellVersion = '5.0'
```

The remaining parameter that we missed will be commented in the manifest module. We can manually edit it using any text editor.

Dynamic modules

A dynamic module is a module that does not persist on the disk but in the memory; so, it will be lost once you close your PowerShell session. This type of modules can be created from the functions and script blocks within the same session, which is useful to developers for a better, object-oriented scripting. It is also useful to administrators at times when they want to execute certain modules on remote computers where they physically exist using PowerShell remoting. Dynamic modules are created using the `New-Module` cmdlet and with parameters such as `-Function` and `-ScriptBlock`, which specify the function and script blocks that are to be included in this module.

In the following example, let's create a dynamic module using the `New-Module` cmdlet:

```
PS C:\> New-Module -ScriptBlock {Function Print {"Dynamic Module Demo!"}; Print}
```

This creates a dynamic module with the exported `Print` cmdlet, as follows:

```
PS C:\> Print
Dynamic Module Demo!
```

Script debugging

Debugging in PowerShell is similar to that in other programming languages. Script debugging is available in the console host as well as in the PowerShell ISE; ISE has both GUI and cmdlet-based debugging, whereas console has only cmdlets to debug your scripts.

The debugging feature in the PowerShell ISE is available under the **Debug** tab, which is self-explanatory. Let's take a look at the breakpoints and debug cmdlets:

```
(Get-Command -Name *Debug*).Where({$_.Source -ne 'Azure' -and  
$_. CommandType -eq 'cmdlet'})  
Get-Command -Name *BreakPoint
```

Note that in the first snippet, I used the `where` method to ignore Azure and to get only the core cmdlets; the output is illustrated in the following image:

PS C:\> (Get-Command -Name *Debug*).Where({\$_.Source -ne 'Azure' -and \$_.CommandType -eq 'cmdlet'})			
CommandType	Name	Version	Source
Cmdlet	Debug-Job	3.0.0.0	Microsoft.PowerShell.Core
Cmdlet	Debug-Process	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Debug-Runspace	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Disable-RunspaceDebug	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Enable-RunspaceDebug	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Get-RunspaceDebug	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Set-PSDebug	3.0.0.0	Microsoft.PowerShell.Core
Cmdlet	Wait-Debugger	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Write-Debug	3.1.0.0	Microsoft.PowerShell.Utility

PS C:\> Get-Command -Name *BreakPoint			
CommandType	Name	Version	Source
Cmdlet	Disable-PSBreakpoint	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Enable-PSBreakpoint	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Get-PSBreakpoint	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Remove-PSBreakpoint	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Set-PSBreakpoint	3.1.0.0	Microsoft.PowerShell.Utility

Managing breakpoints

Windows PowerShell has three types of breakpoints. They are as follows:

Line breakpoints

In this, the script pauses when the designated line is reached during the operation of the script. For example, run the following command:

```
$line = Set-psbreakpoint -script C:\Temp\Scriptdemo.ps1 -Line 4  
$line | fl
```

The output is as follows:

```
Id      : 6  
Script   : C:\Temp\Scriptdemo.ps1  
Line     : 4  
Column   : 0  
Enabled   : True  
HitCount : 0  
Action    :
```

A screenshot of a PowerShell window showing a script. The script contains the following code:

```
$servers = @('localhost' , 'NotOnline' , 'localhost')  
foreach($server in $servers)  
{  
    Get-WmiObject -Class Win32_OperatingSystem -ComputerName $server  
}
```

A red arrow points from a callout bubble to the line `Get-WmiObject -Class Win32_OperatingSystem -ComputerName $server`. The callout bubble contains the text "Line breakpoint enabled".

Variable breakpoints

In this, the script pauses whenever the designated variable's value changes. Run the following command:

```
$Variable = Set-psbreakpoint -script C:\Temp\Scriptdemo.ps1 -  
Variable Server  
$Variable
```

Command breakpoints

In this, the script pauses whenever the designated command is about to be run during the operation of the script. It can include parameters to

further filter the breakpoint to only the operation that you want. The command can also be a function that you create. Run the following command:

```
$cmd = Set-PSBreakpoint -Command Get-Service  
$cmd | Enable-PSBreakpoint
```

The output is illustrated in the following image:

```
PS C:\> Get-PSBreakpoint -id 13  
ID Script Line Command Variable Action  
-- -- -- -- --  
13  
  
PS C:\> Enable-PSBreakpoint -Id 13 -Verbose  
PS C:\> Get-PSBreakpoint -Id 13  
ID Script Line Command Variable Action  
-- -- -- -- --  
13
```

The `Disable-PSBreakpoint` cmdlet disables the breakpoint temporarily, whereas the `Remove-PSBreakpoint` cmdlet removes the breakpoint permanently.

Debugging scripts

After setting up the required breakpoints, we can debug the scripts by simply executing the PS1 file. In this example, we have enabled only the variable breakpoint. Take a look at the following image:

```
PS C:\Temp> .\Scriptdemo.ps1  
Hit Variable breakpoint on 'C:\Temp\Scriptdemo.ps1:$Server' (Write access) 1  
Hit Variable breakpoint on 'C:\Temp\Scriptdemo.ps1:$Server' (Write access)  
[DBG]: PS C:\Temp>> 2  
[DBG]: PS C:\Temp>>  
PS C:\Temp> |
```

The points marked in the figure are explained in the following list:

- 1: Here, the variable breakpoint is hit for `$server`. We see it twice because `$server` is used in two places. To step over use the short cut

key S.

- 2: Now, we are in debug mode.

When we do step over, the script in ISE looks like the following image:

```
$servers = @('localhost' , 'NotOnline' , 'localhost')
foreach($server in $servers)
{
    Get-WmiObject -Class Win32_OperatingSystem -ComputerName $server
```

1

In Windows PowerShell 5.0, `Wait-Debugger` and `Debug-Job` are the two new debugging cmdlets.

- `Debug-Job`: This debugs a running background, remote machine, or Windows PowerShell Workflow job
- `Wait-Debugger`: This stops a script in the debugger before running the next statement in the script, as shown in the following image:

```
PS C:\> $Debug = Start-Job -Name DebugDemo -ScriptBlock {
    $var = 1
    Wait-Debugger
    $var
    $var + 10
}

PS C:\> Debug-Job -Job $Debug -Verbose
VERBOSE: Performing the operation "Debug" on target "DebugDemo".
Stopped at: $var
[DBG]: [Job42]: PS C:\Users\904870\Documents>> |
```

In the following example, we have used the `Set-PSBreakPoint` cmdlet instead of the `Wait-Debugger` cmdlet:

```
$job = Start-Job -ScriptBlock { Set-PSBreakpoint
C:\Temp\Scriptdemo.ps1 -Line 4; C:\Temp\Scriptdemo.ps1 }
$job
```

PS C:\> Get-Job						
Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
53	Job53	BackgroundJob	AtBreakpoint	True	localhost	Set-PSBreakpoint C:\T...

In the figure we can see that the state is AtBreakpoint.

The following cmdlet will enter debug mode of the running job:

```
Debug-Job $job -Verbose
```

Windows PowerShell 5.0 makes it easy for us to debug the PowerShell scripts. Using the Set-PSBreakPoint cmdlet, we can pause the execution of the script at a particular line before we proceed further. Just to avoid the risk! Debugging is easy in Windows PowerShell!

Summary

Windows PowerShell 5.0 preview has got many more significant features, such as enhancement in PowerShell DSC, new cmdlets and improvements in existing cmdlets, the ISE supporting transcriptions, supporting Class, the ability to create Custom DSC resources and easy string manipulation using Class, the introduction of the new Network Switch module that helps automate and manage Microsoft-signed network switches, and so on.

In this chapter, we discussed a few great features of Windows PowerShell, such as exploring the basics of DSC, writing classes, PowerShell debugging, and so on.

In the next chapter, we will cover Windows PowerShell 5.0 Desired State Configuration, Remote Management, and more.

Chapter 3. Exploring Desired State Configuration

In the previous chapter, we covered the basics of Desired State Configuration; this chapter is a continuation of the same.

To learn DSC, you would need to watch the video in Microsoft Virtual Academy.

Note

Here are some useful links:

To get started with Windows PowerShell Desired State Configuration (DSC)—<http://www.microsoftvirtualacademy.com/training-courses/getting-started-with-powershell-desired-state-configuration-dsc->

For advanced PowerShell Desired State Configuration (DSC) and custom resources—<http://www.microsoftvirtualacademy.com/training-courses/advanced-powershell-desired-state-configuration-dsc-and-custom-resources>

DSC—Desired State Configuration—is a Windows PowerShell extension released with Windows Management Framework 4.0. DSC is a fast-growing technology. Any IT professional or developer responsible for configuration management automation needs DSC. In this chapter, we will cover the following topics:

- The prerequisites for DSC
- Installing WMF 5.0 using DSC
- Imperative versus declarative programming
- Getting started with DSC
- Understanding MOF
- Exploring WinRM and CIM
- Creating and deploying a configuration
- Types of deployment modes

Prerequisites

DSC needs a supporting PowerShell version, which is version 4.0, and operating system from Windows 2008 R2 onwards. DSC is built on CIM and needs the WinRM listeners, service, and script execution policy.

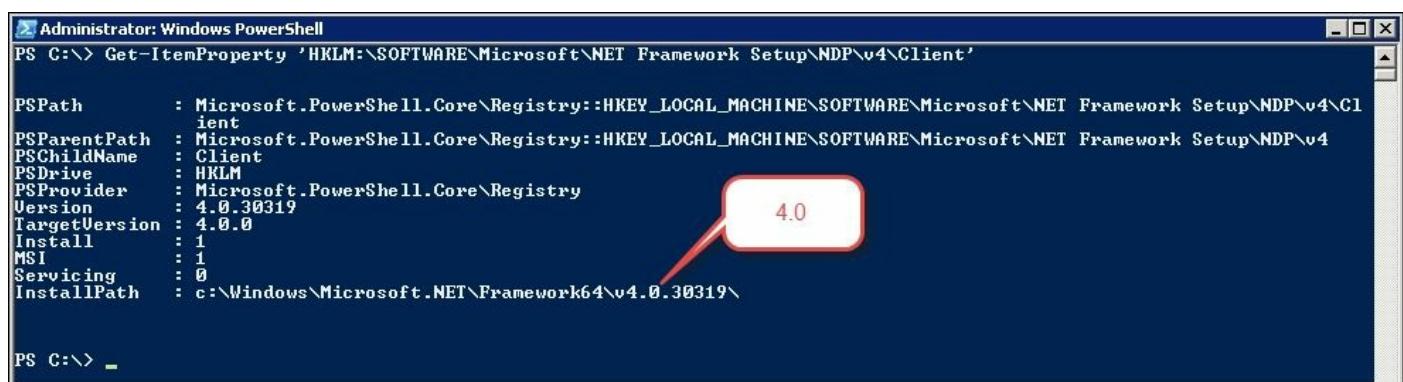
Let's set up a machine to jump-start DSC. Here, we will use a machine working on the Windows Server 2008 R2 operating system with SP1, and the PowerShell version is 2.0. Perform the following steps:

- Download .NET Framework 4.5 using the URL:
<http://www.microsoft.com/en-us/download/details.aspx?id=40855>
- Download WMF 4.0 from the following link:
<http://www.microsoft.com/en-us/download/details.aspx?id=40855>

Let's take a look at the .NET framework version installed on the Windows 2008 R2 SP1 box from the registry using PowerShell. Run the following command for this:

```
Get-ItemProperty 'HKLM:\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Client'
```

The output is illustrated in the following image:



```
Administrator: Windows PowerShell
PS C:\> Get-ItemProperty 'HKLM:\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Client'

PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Client
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4
PSChildName : Client
PSDrive     : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
Version     : 4.0.30319
TargetVersion: 4.0.0
Install     : 1
MSI         : 1
Servicing   : 0
InstallPath  : c:\Windows\Microsoft.NET\Framework64\v4.0.30319\
```

A red callout bubble points to the 'Version' property value '4.0'.

You can also install the .NET Framework 4.5 first and then check the version after successful upgrade of .NET Framework by executing the code mentioned before, which shows the version as 4.5.50709. The machine we will use for this demo doesn't have Windows PowerShell

ISE installed, so let's enable the feature using the following PowerShell code:

```
Import-Module ServerManager -Verbose  
Get-WindowsFeature -Name '*ISE*' -verbose  
Add-WindowsFeature -Name 'PowerShell-ISE' -Verbose
```

Yeah! It's done. Take a look at the following image:

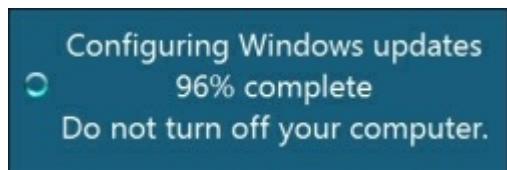
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history and output are as follows:

```
PS C:\> Import-Module ServerManager -Verbose 1  
VERBOSE: Loading module from path  
'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\ServerManager\ServerManager.psd1'.  
VERBOSE: Loading 'FormatsToProcess' from path  
'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\ServerManager\Feature.format.ps1xml'.  
VERBOSE: Importing cmdlet 'Get-WindowsFeature'.  
VERBOSE: Importing cmdlet 'Add-WindowsFeature'.  
VERBOSE: Importing cmdlet 'Remove-WindowsFeature'.  
VERBOSE: Exporting cmdlet 'Get-WindowsFeature'.  
VERBOSE: Exporting cmdlet 'Add-WindowsFeature'.  
VERBOSE: Exporting cmdlet 'Remove-WindowsFeature'.  
VERBOSE: Importing cmdlet 'Add-WindowsFeature'.  
VERBOSE: Importing cmdlet 'Get-WindowsFeature'.  
VERBOSE: Importing cmdlet 'Remove-WindowsFeature'.  
PS C:\> Get-WindowsFeature -Name '*ISE*' 2  
Display Name Name  
[+] Windows PowerShell Integrated Scripting Environment PowerShell-ISE  
3  
PS C:\> Add-WindowsFeature -Name 'PowerShell-ISE' -Verbose 4  
VERBOSE: Checking if running in 'WhatIf' Mode.  
VERBOSE: Start Installation...  
VERBOSE: Performing operation "Add-WindowsFeature" on Target "[Windows PowerShell Integrated Scripting Environment (ISE)] Windows PowerShell Integrated Scripting Environment (ISE)".  
VERBOSE: Performing operation "Add-WindowsFeature" on Target "[.NET Framework 3.5.1 Features] .NET Framework 3.5.1".  
VERBOSE: [Installation] Succeeded: [.NET Framework 3.5.1 Features] .NET Framework 3.5.1.  
VERBOSE: [Installation] Succeeded: [Windows PowerShell Integrated Scripting Environment (ISE)] Windows PowerShell Integrated Scripting Environment (ISE).  
Success Restart Needed Exit Code Feature Result  
True 5 No Success <.NET Framework 3.5.1, Windows PowerShell ...  
PS C:\> Get-WindowsFeature -Name '*ISE*'  
Display Name Name  
[+] Windows PowerShell Integrated Scripting Environment PowerShell-ISE
```

Annotations with red circles and numbers:

- Annotation 1: Points to the first line of the "Import-Module" command.
- Annotation 2: Points to the "Get-WindowsFeature" command.
- Annotation 3: Points to the "Windows PowerShell Integrated Scripting Environment" entry in the table.
- Annotation 4: Points to the first line of the "Add-WindowsFeature" command.
- Annotation 5: Points to the "Success" status in the "Feature Result" column.

Install the WMF 4.0 MSU file; you can refer to the steps used for this in [Chapter 1, Getting Started with Windows PowerShell](#). This installation requires a reboot, so plan according to your production environment. Take a look at the following image:



After the reboot, PS will be upgraded to version 4.0, and we can take a

look at the `PSDesiredStateConfiguration` module, as shown in the following image:

```
PS C:\> Get-Module -ListAvailable

Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules

ModuleType Version Name                                ExportedCommands
---- -- -- -- -- --
Manifest   1.0.0.0 ADRMS
Manifest   1.0.0.0 AppLocker
Manifest   1.0 BestPractices
Manifest   1.0.0.0 BitsTransfer
Manifest   1.0.0.0 CimCmdlets
Script     1.0.0.0 ISE
Manifest   3.0.0.0 Microsoft.PowerShell.Diagnostics
Manifest   3.0.0.0 Microsoft.PowerShell.Host
Manifest   3.1.0.0 Microsoft.PowerShell.Management
Manifest   3.0.0.0 Microsoft.PowerShell.Security
Manifest   3.1.0.0 Microsoft.PowerShell.Utility
Manifest   3.0.0.0 Microsoft.WSMAN.Management
Binary     1.0 PSDesiredStateConfiguration
Script     1.0.0.0 PSDiagnostics
Binary     1.1.0.0 PSScheduledJob
Manifest   2.0.0.0 PSWorkflow
Manifest   1.0.0.0 PSWorkflowUtility
Manifest   1.0.0.0 ServerManager
Manifest   1.0.0.0 TroubleshootingPack

PSDesiredStateConfiguration has a red arrow pointing to it.
```

We have now successfully installed Windows Management Framework 4.0; in the next chapter, we will discuss the steps to upgrade WMF to version 5.0, the April 2015 preview, using DSC.

Installing the WMF 5.0 April 2015 preview

The latest released version of WMF 5.0 is the April 2015 preview. It's good to read the release document, refer the following link:

<https://www.microsoft.com/en-us/download/details.aspx?id=46889>

The PowerShell team releases experimental DSC resources to configure an environment, and the latest DSC resource kit is **Wave 10**. Let's download this from the **TechNet Gallery** website (<https://gallery.technet.microsoft.com/scriptcenter/DSC-Resource-Kit-All-c449312d>) to install WMF 5.0 because we need the **xHotfix** resource to perform this action. We will discuss all about the DSC resources later in this chapter.

Note that from April 2015 onward, the DSC resource kit has been outsourced to **GitHub**.

The central repository for DSC resources can be found at <https://github.com/powershell/DscResources>.

In this example, we will install the MSU file for WMF 5.0 using DSC; to do this, we will use the `Configuration` keyword with the following script block:

```
Configuration WMF5
{
    Import-DscResource -ModuleName xWindowsUpdate
    Node localhost
    {
        xHotfix WMFInstall
        {
            Path = 'C:\Users\ChenV\Downloads\Windows6.1-KB2908075-x64.msu'
            Id = 'KB2908075'
            Ensure = 'Present'
        }
    }
}
```

}

WMF5

How does this work? Take a look at the following image:

The screenshot shows a PowerShell session in progress. The command entered is:

```
PS C:\> Configuration WMF5
{
    Import-DscResource -ModuleName xWindowsUpdate, xPendingReboot 1
    Node localhost 2
    {
        xHotfix WMFIInstall 3
        {
            Path = 'C:\Users\ChenV\Downloads\Windows6.1-KB2908075-x64.msu' 4
            Id = 'KB2908075'
            Ensure = 'Present'
        }
    }
} 5
WMF5 6
```

After the command is run, a message box appears with the text "MOF Created". Below the command output, a table shows the file details:

Mode	LastWriteTime	Length	Name
-a---	7/2/2015 9:20 AM	1192	localhost.mof

The points marked in the figure are explained in the following list:

- **1:** Here, we use the `Configuration` keyword and keyword and block.
- **2:** Here, we import the experimental DSC resource, `xWindowsUpdate`. `Import-DscResource` should be inside the configuration block.
- **3:** Here, we declare the node. We will use the `localhost`.
- **4:** Here, we use the `xHotfix` DSC resource and give it a friendly name.
- **5:** Here, we declare the DSC properties—`Path`, `ID`, and `Ensure`.
- **6:** Here, we call the configuration to create an MOF file. As you did with the function, use a friendly name.
- **MOF Created:** Here, we have successfully created our MOF file.

After creating the MOF file, we can apply the configuration using `Start-DscConfiguration`. Wait! First, let's do some additional setting up, such as updating the help, execution policy, WinRM services, and so on. Take a look at the following code. It will update the help files, modify

the execution policy to `RemoteSigned`, and `Set-WSManQuickConfig` in it will do the following:

- It will check whether the WinRM service is running. If it is not running, the service will be started.
- It will set the WinRM service startup type to automatic.
- It will create a listener that will accept requests on any IP address. By default, the transport is HTTP.
- It will enable a firewall exception for the WinRM traffic.

By default, WinRM is enabled in Windows Server 2012 and Windows 8.1. We will perform this action here because we will be using Windows Server 2008 R2 SP1.

`Start-DscConfiguration` will apply the configuration on the target node, which is our localhost. Run the following command:

```
Update-Help -Verbose  
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Verbose  
Set-WSManQuickConfig -Verbose  
Start-DscConfiguration .\WMF5 -Wait -Force -WhatIf
```

Let's take a look at the output of `Start-DscConfiguration`:

```
PS C:\> Start-DscConfiguration .\WMF5 -Wait -Force -Verbose  
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters: {'methodName' = SendConfigurationApply,'className' = MSFT_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft  
/Windows/DesiredStateConfiguration}  
VERBOSE: An LCM method call arrived from computer DSC2008R2DEMO with user sid S-1-5-21-696186302-4281343859-1456728110-500.  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Set ]  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Resource ] [[xHotfix]\WMPInstall]  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Test ] ① [[xHotfix]\WMPInstall]  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Test ] [[xHotfix]\WMPInstall] Testing whether hotfix is Present ②  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Test ] [[xHotfix]\WMPInstall] Validating path/URL. ③  
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Test ] [[xHotfix]\WMPInstall] in 5.6240 seconds.  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Set ]  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Resource ] [[xHotfix]\WMPInstall]  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Set ] [[xHotfix]\WMPInstall] Log name hasn't been specified. Hotfix will use the temporary log C:\Windows\TEMP\tmpF966.tmp.et ⑤  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Set ] [[xHotfix]\WMPInstall] Validating path/URL.  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Set ] [[xHotfix]\WMPInstall] START Install using wsusa.exe ⑥  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Set ] [[xHotfix]\WMPInstall] Found file c:\windows\temp\tmpf966.tmp.et  
VERBOSE: [DSC2008R2DEMO]: LCM: [ Set ] [[xHotfix]\WMPInstall] END Install using wsusa.exe ⑦  
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Set ] [[xHotfix]\WMPInstall] in 80.8020 seconds.  
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Set ] in 89.8640 seconds.  
VERBOSE: Operation 'Invoke CimMethod' complete. ⑧  
VERBOSE: Time taken for configuration job to complete is 92.163 seconds
```

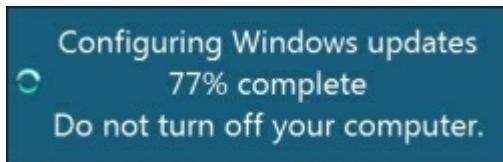
The points marked in the figure are explained in the following list:

- 1: Here, the configuration is started
- 2: Here, a test is performed to check whether the hotfix is present
- 3: Here, the path is validated
- 4: Here, it shows the time consumed for validation
- 5: Here, since we haven't provided the log path, DSC creates a log

file in C:\Windows\Temp

- 6: Here, the MSU installation happens using wsusa.exe
- 7: Here, the installation is completed
- 8: Here, it shows the total time consumed to apply the configuration

After a reboot, we can see that the updates are being installed, as in the following image:



We have applied the configuration successfully; the box is in WMF 5.0, as shown in the following image:

```
PS C:\> $PSVersionTable
Name          Value
----          -----
PSVersion     5.0.10105.0
WSManStackVersion 3.0
SerializationVersion 1.1.0.1
CLRVersion    4.0.30319.17929
BuildVersion   10.0.10105.0
PSCompatibleVersions {1.0, 2.0, 3.0, 4.0...}
PSRemotingProtocolVersion 2.3
```

A red speech bubble points from the right towards the "Value" column of the PSVersion table entry. Inside the bubble, the word "Success!" is written in red text.

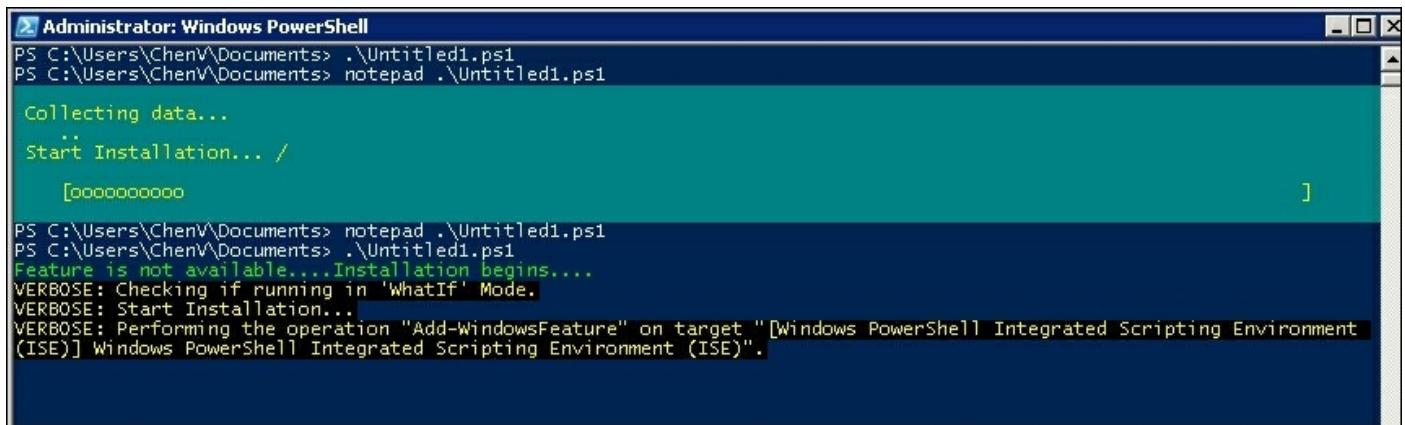
Imperative versus declarative programming

Imperative programming means describing the computations in terms of statements, as we do in procedural programming. In imperative programming, we instruct the computer on which operation to do and how to do it. It is an explicit sort of coding. For example, we use imperative programming to perform certain operations and catch all the exceptions that may occur, or to install software if it does not exist on a specific machine. PowerShell uses imperative programming style, except in the DSC feature. In the preceding topic, we installed the PowerShell ISE using the `ServerManager` module, which is an imperative programming practice. Let's revisit it—what we will do here is check whether the PowerShell ISE feature is installed; if not, we will proceed with the installation.

We removed the PowerShell ISE using the `Remove-WindowsFeature` cmdlet. Now, we will execute the following code to install the PowerShell ISE:

```
Import-Module ServerManager
If (-not (Get-WindowsFeature -Name "PowerShell-ISE").Installed)
{
    try {
        Write-Host "Feature is not available....Installation begins...." -ForegroundColor Green
        Add-WindowsFeature -Name "PowerShell-ISE" -ErrorAction Stop -Verbose
    }
    catch {
        $_.Exception
    }
}
```

How does this work? Take a look at the following image:



```
Administrator: Windows PowerShell
PS C:\Users\Chen\Documents> .\Untitled1.ps1
PS C:\Users\Chen\Documents> notepad .\Untitled1.ps1
Collecting data...
Start Installation... /
[oooooooooooo] ]]

PS C:\Users\Chen\Documents> notepad .\Untitled1.ps1
PS C:\Users\Chen\Documents> .\Untitled1.ps1
Feature is not available....Installation begins...
VERBOSE: Checking if running in 'WhatIf' Mode.
VERBOSE: Start Installation...
VERBOSE: Performing the operation "Add-WindowsFeature" on target "[Windows PowerShell Integrated Scripting Environment (ISE)] Windows PowerShell Integrated Scripting Environment (ISE)".
```

- It imports the ServerManager module
- It checks the installation status; if this is true, no action is taken
- If the installation status returns false, the installation will begin
- Adding the PowerShell ISE feature doesn't need a reboot, but removing it requires one.

Let's take a look at how the same operation can be performed using PowerShell DSC.

In PowerShell DSC, we can do this using declarative syntax. For this, we will use the Configuration block, as shown in the following code:

```
Configuration InstallISE
{
    Node localhost
    {
        WindowsFeature ISE
        {
            Name = 'PowerShell-ISE'
            IncludeAllSubFeature = $true
            LogPath = 'C:\LogISE.txt'
            Ensure = 'Present'
        }
    }
}
InstallISE
```

Take a look at the verbose message in the following image—here, it skipped the installation because the feature is already installed:

```

PS C:\Users\ChenV> Configuration InstallISE
{
    Node $ENV:COMPUTERNAME
    {
        WindowsFeature ISE
        {
            Name = 'PowerShell-ISE'
            IncludeAllSubFeature = $true
            LogPath = 'C:\LogISE.txt'
            Ensure = 'Present'
        }
    }
}

InstallISE

WARNING: The configuration 'InstallISE' is loading one or more built-in resources without explicitly importing associated modules. Add Import-DscResource -Module 'PSDesiredStateConfiguration' to your configuration to avoid this message.

Directory: C:\Users\ChenV\InstallISE

Mode          LastWriteTime     Length Name
----          -----      1988 DSC2008R2DEMO.mof

PS C:\Users\ChenV> Start-DscConfiguration .\InstallISE -Wait -Force -Verbose
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, {"methodName" = SendConfigurationApply, "className" = MSFT_DSCLocalConfigurationManager, "namespaceName" = root/Microsoft/Windows/DesiredstateConfiguration}.
VERBOSE: An LCM method call arrived from computer DSC2008R2DEMO with user sid S-1-5-21-696186302-4281343859-1456728110-500.
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Resource ] Skipped
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Resource ] [[WindowsFeature]ISE]
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Test ] [[WindowsFeature]ISE]
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Test ] [[WindowsFeature]ISE] in 1.6560 seconds.
VERBOSE: [DSC2008R2DEMO]: LCM: [ Skip Set ] [[WindowsFeature]ISE]
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Resource ] [[WindowsFeature]ISE]
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Set ] in 3.2500 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.

```

By simply changing the `Ensure` property to `Absent`, we can uninstall the feature, as shown in the following code:

```

Configuration InstallISE
{
    Node $ENV:COMPUTERNAME
    {
        WindowsFeature ISE
        {
            Name = 'PowerShell-ISE'
            IncludeAllSubFeature = $true
            LogPath = 'C:\LogISE.txt'
            Ensure = 'Absent'
        }
    }
}

InstallISE

```

Take a look at the verbose message, which appears while applying the configuration, in the following image; it informs us about the reboot, so the action is not completed until we reboot the machine:

```

PS C:\Users\Chen> Start-DscConfiguration \\InstallISE -Wait -Force -Verbose
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, 'methodName' = SendConfigurationApply,'className' = MSFT_DSCLocalConfigurationManager,'namespaceName' =
root\Microsoft\Windows\DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer DSC2008R2DEMO with user sid S-1-5-21-696186302-4281343859-1456728110-500.
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Set ] {[WindowsFeature]ISE}
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Resource ] {[WindowsFeature]ISE}
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Test ] {[WindowsFeature]ISE} in 1.5940 seconds.
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Test ] {[WindowsFeature]ISE}
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Set ] {[WindowsFeature]ISE}
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Set ] {[WindowsFeature]ISE} Checking if running in 'WhatIf' Mode.
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Set ] {[WindowsFeature]ISE} Start Removal...
VERBOSE: [DSC2008R2DEMO]: LCM: [ Start Set ] {[WindowsFeature]ISE} Performing the operation "Remove-WindowsFeature" on target "[Windows PowerShell Integrated Scripting Environment (ISE)] Windows PowerShell Integrated Scripting Environment (ISE)".
WARNING: [DSC2008R2DEMO]: LCM: [ Start Set ] {[WindowsFeature]ISE} [Removal] Succeeded: [Windows PowerShell Integrated Scripting Environment (ISE)] Windows PowerShell Integrated Scripting Environment (ISE). You must restart this server to finish the removal process.
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Set ] {[WindowsFeature]ISE} Successfully uninstalled the feature PowerShell-ISE.
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Set ] {[WindowsFeature]ISE} The Target machine needs to be restarted.
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Set ] {[WindowsFeature]ISE} in 11.3120 seconds.
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Set ] {[WindowsFeature]ISE} A reboot is required to progress further. Please reboot the system.
WARNING: [DSC2008R2DEMO]: LCM: [ End Set ] {[WindowsFeature]ISE} A reboot is required to progress further. Please reboot the system.
VERBOSE: [DSC2008R2DEMO]: LCM: [ End Set ] {[WindowsFeature]ISE} in 13.6710 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 13.815 seconds

```

Here is a comparison of imperative and declarative programming:

Imperative	Declarative
This describes the task in statements.	This describes the action required.
A step-by-step procedure of execution is required in the code.	In this, we describe the requirement in the code, and the machine executes the task as required.

Let's review a few more declarative samples to study DSC's features and benefits.

Getting started with DSC

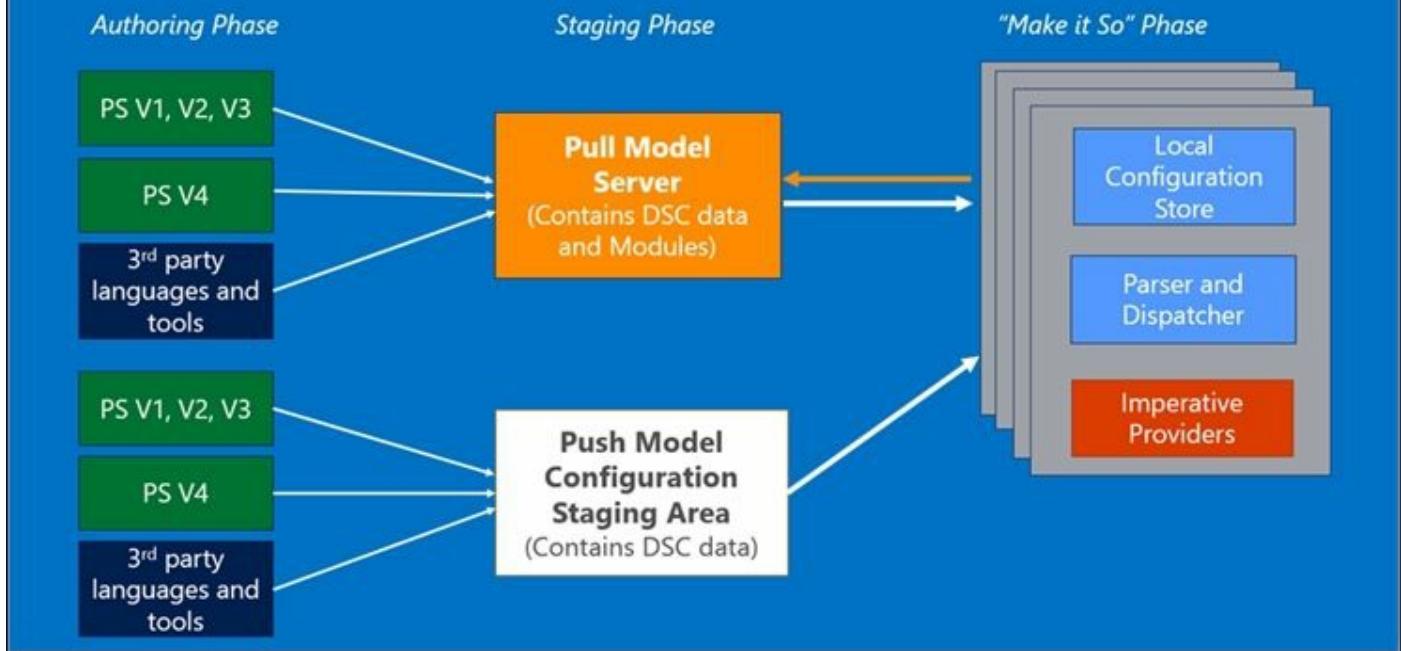
Using DSC, we can configure our environment through codes. As we know, PowerShell DSC is not a tool; it is a configuration management platform. DSC uses a `Configuration` keyword. If you are an IT professional or developer responsible for infrastructure compliance and automation, DSC will help you perform most of your tasks easily and swiftly.

DSC enables a consistent infrastructure, standardized configuration, and continuous deployments. Using DSC, we can remediate the drift in our environment. Following are a few of the most common use cases of DSC:

- Enabling or disabling server roles and features
- Managing registry settings
- Managing files and directories
- Starting, stopping, and managing processes and services
- Managing groups and user accounts
- Deploying new software
- Managing environment variables
- Running Windows PowerShell scripts
- Fixing a configuration that has drifted away from the desired state
- Discovering the actual configuration state on a given node

The architecture of DSC is illustrated in the following image:

PowerShell DSC – Pull/Push Model



Now, it's time for us to explore all the three phases of DSC shown in the preceding image. This image is taken from <http://blogs.technet.com/b/privatecloud/archive/2013/08/30/introducing-powershell-desired-state-configuration-dsc.aspx>.

The Authoring phase

DSC can be created using PowerShell or any other third-party extensions. We will stick to PowerShell because that's our scope. This is the phase where the configuration is authored using declarative scripting.

Configuration is the core component of DSC, and the base structure looks similar to the following:

```
Configuration <Identifier>
{
}
```

Here, `Configuration` is the keyword and `<Identifier>` is the string (or

friendly name) of our configuration, followed by the {} block. Inside the configuration block, we will use `Node` and the resources as illustrated in the following code:

```
Configuration Test
{
    Node localhost
    {
        File Test
        {
        }
    }
}
```

Here, `Node` is the target node, and `File` is the built-in resource used to manage the file or directories.

Let's switch to Windows Server 2012 to continue the demo; there are a few differences in this, but the PS version remains the same. Execute the following code:

```
Configuration Test
{
    Node localhost
    {
        File Test
        {
            Ensure = 'Present'
            DestinationPath = 'C:\'
            Contents = 'PowerShell DSC'
        }
    }
}
Test
```

The configuration script creates a folder as `Test` (name of the configuration we created) in `C:\` and saves the MOF file over there. Take a look at the following image, which shows the `Test` folder and the `localhost.mof` file:

Directory: C:\Test		
Mode	LastWriteTime	Length Name
-a----	8/24/2015 12:03 PM	1892 localhost.mof

The Staging phase

Once the configuration script is created and executed, the next phase is staging, which is nothing but generating the MOF file. We haven't discussed the Push and Pull modes yet, so we will save the MOF file in a local machine. The MOF file can be saved in a central location, **Server Message Block (SMB)**, with the appropriate access to write MOF.

The preceding image shows the generated MOF; this is not enacted yet. This is a staging phase.

The "Make it so" phase

The enactment of the configuration takes place in this phase, and then the configuration script applies the changes to the target nodes. The configuration is applied on the target node using **Local Configuration Manager (LCM)**, which is an engine that comes with WMF versions from 4.0.

In PowerShell 5.0, the LCM version is 2.0, which is significantly improved. To see the meta configuration of LCM, we can execute the following codes; both return the same result:

```
Get-CimClass -ClassName MSFT_DscMetaConfiguration -Namespace root/microsoft/Windows/DesiredStateConfiguration | Select -ExpandProperty CimClassProperties | Select Name
Get-DscLocalConfigurationManager | GM -MemberType Properties | Select Name
```

Now, let's push the configuration using the `Start-DscConfiguration` cmdlet, and take a look at the LCM state. This enacts the configuration

in the localhost and changes are effected immediately, as shown in the following image:

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". On the left, a red callout box points to the text "Configuring Web Server!". On the right, another red callout box points to the text "Enacting Configuration!". The main window displays DSC configuration logs and the output of the command "Get-DscLocalConfigurationManager | Select LCMState".

```
Start Installation....  
68%, 00:00:00 remaining, [DSCDEMOLAB] Set-TargetResource.  
  
VERBOSE: An LCM method was called on computer DSCDEMOLAB with user sid S-1-5-21-1526373039-4077562782-362964726  
VERBOSE: [DSCDEMOLAB]: [Resource] [[File]DSCConfig]  
VERBOSE: [DSCDEMOLAB]: [Resource] [[File]DSCConfig]  
VERBOSE: [DSCDEMOLAB]: [Resource] [[File]DSCConfig] The destination object was found and no action  
VERBOSE: [DSCDEMOLAB]: LCM: [ End Test ]  
VERBOSE: [DSCDEMOLAB]: LCM: [ Skip Set ]  
VERBOSE: [DSCDEMOLAB]: LCM: [ End Resource ]  
VERBOSE: [DSCDEMOLAB]: LCM: [ Start Resource ]  
VERBOSE: [DSCDEMOLAB]: LCM: [ Start Test ]  
VERBOSE: [DSCDEMOLAB]:  
VERBOSE: [DSCDEMOLAB]:  
VERBOSE: [DSCDEMOLAB]: LCM: [ End Test ]  
VERBOSE: [DSCDEMOLAB]: LCM: [ Start Set ]  
VERBOSE: [DSCDEMOLAB]:  
VERBOSE: [DSCDEMOLAB]:  
VERBOSE: [DSCDEMOLAB]:  
VERBOSE: [DSCDEMOLAB]:  
VERBOSE: [DSCDEMOLAB]:  
  
Windows PowerShell  
Copyright (C) 2015 Microsoft Corporation. All rights reserved.  
PS C:\Users\ChenV> Get-DscLocalConfigurationManager | Select LCMState  
  
LCMState  
-----  
Busy  
  
PS C:\Users\ChenV>
```

Local Configuration Manager

Local Configuration Manager is the engine of Windows PowerShell DSC. The configurations we push to the nodes are parsed using LCM, and it gets the required resources to keep the machine in the desired state.

To modify the LCM settings, you can use the following code:

```
[DscLocalConfigurationManager ()]  
Configuration LCM  
{  
    Node localhost  
    {  
        Settings  
        {  
            RebootNodeIfNeeded = $true  
        }  
    }  
}  
LCM
```

In the preceding example, we enacted the configuration using `Start-DscConfiguration`, but to apply the meta configuration changes, we

need to use the `Set-DscLocalConfigurationManager` cmdlet.

If we execute the preceding code, it would create an MOF file such as `nodeName.Meta.MOF`. Running the following command changes the LCM settings:

```
PS C:\> Set-DscLocalConfigurationManager .\LCM -Verbose
VERBOSE: Performing the operation "Start-DscConfiguration" on target "MSFT_DSCLocalConfigurationManager".
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, {'methodName' = SendMetaConfigurationApply, 'className' = MSFT_DSCLocalConfigurationManager, 'namespaceName' = root\Microsoft\Windows\DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer DSCDEMOLAB with user sid S-1-5-21-1526373039-4077562782-3629647265-500.
VERBOSE: [DSCDEMOLAB]: LCM: [ Start Set ]
VERBOSE: [DSCDEMOLAB]: LCM: [ Start Resource ] [MSFT_DSCLocalConfiguration]
VERBOSE: [DSCDEMOLAB]: LCM: [ Start Set ] [MSFT_DSCLocalConfiguration]
VERBOSE: [DSCDEMOLAB]: LCM: [ End Set ] [MSFT_DSCLocalConfiguration] in 0.0160 seconds.
VERBOSE: [DSCDEMOLAB]: LCM: [ End Resource ] [MSFT_DSCLocalConfiguration]
VERBOSE: [DSCDEMOLAB]: LCM: [ End Set ]
VERBOSE: [DSCDEMOLAB]: LCM: [ End Set ] in 0.0620 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Set-DscLocalConfigurationManager finished in 0.274 seconds.
```

Parameterizing the configuration script

For now, we have a basic understanding of DSC in PowerShell. Let's discuss parameterizing the configuration code. The following code is just an example, where we will create a folder in `C:\Temp` or whichever path is passed as the parameter value:

```
Configuration CreateFolder
{
    param([Parameter(Mandatory = $true)]
        $FolderName,
        [Parameter(Mandatory = $true)]
        $ComputerName,
        $Ensure = 'Present',
        $Type = 'Directory'
    )
    Node $ComputerName
    {
        File CreateFolder
        {
            DestinationPath =  $FolderName
            Ensure = $Ensure
            Type = $Type
        }
    }
}
```

```
CreateFolder -FolderName 'C:\Temp\PSDSC' -ComputerName  
localhost  
Start-DscConfiguration .\CreateFolder -Wait -Verbose -Force
```

Really, this is cool stuff! We can do many more things easily using parameterizing which allows us to make the configuration script more dynamic. Execute the following code:

```
Configuration WebServer  
{  
    param([string[]]$NodeName)  
    Node $NodeName  
    {  
        WindowsFeature WebServer  
        {  
            Name = "Web-Server"  
            Ensure = "Present"  
            IncludeAllSubFeature = $true  
        }  
    }  
}  
WebServer -NodeName 'WMF05Node4' , 'WMF05Node5'
```

The preceding code generates two MOF files for their respective servers.

Understanding MOF

MOF stands for **Managed Object Format**. These files are compiled using `mofcomp.exe`. A WMI provider normally consists of an MOF file, which defines the data and the event classes for which the provider returns the data, and a DLL file, which contains the code that supplies the data.

To take a look at the MOF structure, we can execute the following code:

```
([wmiclass]"Win32_OperatingSystem").GetText("mof")
```

This returns the MOF structure of the `Win32_OperatingSystem` class.

Using MOF, we can perform tasks such as the following:

- Compiling MOF files
- Creating classes, instances, methods, and comments
- Adding a qualifier

Note

For more information about MOF and its supported operations, refer to the following link:

<https://msdn.microsoft.com/en-us/library/aa823192%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396>

So, using DSC, when we execute the following code, an MOF file with the name `localhost` will be created:

```
Configuration TestMOF
{
    Node Localhost
    {
        Service BITS
        {
            Name = 'BITS'
            State = 'Stopped'
            StartupType = 'Automatic'
        }
    }
}
TestMof
/
@TargetNode='localhost'
@GeneratedBy=ChenV
@GenerationDate=08/26/2015 15:33:19
@GenerationHost=ComputerName
*/
instance of MSFT_ServiceResource as $MSFT_ServiceResource1ref
{
    ResourceID = "[Service]BITS";
    State = "Stopped";
    SourceInfo = "::5::9::Service";
    Name = "BITS";
    StartupType = "Automatic";
    ModuleName = "PsDesiredStateConfiguration";
}
```

```
ModuleVersion = "0.0";
ConfigurationName = "TestMOF";
};

instance of OMI_ConfigurationDocument
{
    Version="2.0.0";
        MinimumCompatibleVersion = "1.0.0";
        CompatibleVersionAdditionalProperties=
    {"Omi_BaseResource:ConfigurationName"};
        Author="ChenV";
        GenerationDate="08/26/2015 15:33:19";
        GenerationHost="ChenVDSC";
        Name="TestMOF";
};

```

If you are familiar with MOF, you can create an MOF file without using PowerShell.

Since we have an MOF file, which leads to BITS being in the `Stopped` state, we can simply edit the MOF file and get this status to `Running`; for example, we can set `State` from `State = 'Stopped'` to be changed to `State = 'Running'`.

Exploring Windows Remote Management and CIM

Every IT professional is interested in managing their servers remotely. It would be better if we can achieve our tasks on remote machines from our desk than having to travel to a data center (unless there's a physical failure).

Remoting is the key feature of Windows PowerShell. However, most security administrators consider this to be a security risk. This is not completely true; Windows PowerShell remoting works based on a two-way authentication (a mutual authentication), and this inherits the feature of the Active Directory Kerberos protocol. This applies only for domain-joined machines.

We will cover the following topics:

- Understanding WS-Management cmdlets
- The HTTP listener
- The HTTPS listener
- Exploring the CIM commands
- Exploring the CIM methods
- Querying the remote machines using CIM

Windows PowerShell remoting

The Windows PowerShell remoting feature makes it richer, and this works based on the WS-Management protocol and the **Windows Remote Management (WinRM)** service. From Windows PowerShell 3.0 onward, the WS-Management configurations can be manipulated using the PowerShell provider, which is the WSMAN drive.

WS-Management is a SOAP-based protocol for servers and is according to the DMTF open-based standard. Following are the standard abilities of DMTF **Web Services Management (WSMan)**:

- It gets, puts (updates), creates, and deletes individual resource instances, such as settings and dynamic values
- It enumerates the contents of the containers and collections, such as large tables and logs
- It subscribes to the events emitted by the managed resources
- It executes the specific management methods with strongly typed input and output parameters

The WinRM service processes the request sent by WSMAN over the network and the listening happens through the `HTTP.sys` driver.

Exploring WSMAN cmdlets

WSMAN cmdlets are used to manage WSMAN protocols. These commands are organized in the `Microsoft.WSMAN.Management` module. Run the following command:

```
Get-Command -Module Microsoft.WSMAN.Management
```

The output is as shown in the following image:

PS C:\> Get-Command -Module Microsoft.WSMAN.Management			
CommandType	Name	Version	Source
Cmdlet	Connect-WSMAN	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Disable-WSMANCredSSP	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Disconnect-WSMAN	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Enable-WSMANCredSSP	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Get-WSMANCredSSP	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Get-WSMANInstance	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Invoke-WSMANAction	3.0.0.0	Microsoft.WSMAN....
Cmdlet	New-WSMANInstance	3.0.0.0	Microsoft.WSMAN....
Cmdlet	New-WSMANSessionOption	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Remove-WSMANInstance	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Set-WSMANInstance	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Set-WSMANQuickConfig	3.0.0.0	Microsoft.WSMAN....
Cmdlet	Test-WSMAN	3.0.0.0	Microsoft.WSMAN....



Note

For more information, refer to the following link:

<https://technet.microsoft.com/en-us/library/hh849876.aspx>

In this exercise, let's take a look at how we can change the maximum session configurations. Let's consider the current maximum allowed connections.

Using the native PowerShell `Get-Item` command, execute the following code and refer to the following image:

```
CD WSMAN:\WMF5Node03\Shell\MaxShellsPerUser
```

The screenshot shows a PowerShell session with the following steps highlighted:

- PS C:\> cd WSMAN: 1
- PS WSMAN:\> dir 2
- PS WSMAN:\wmf5node03> cd .\Shell 3
- PS WSMAN:\wmf5node03\Shell> dir 4

A red callout bubble points from step 1 to the "WSManConfig:" object in the first table. Another red callout bubble points from step 4 to the "MaxShellsPerUser" row in the second table.

Type	Name	SourceOfValue	Value
System.String	AllowRemoteShellAccess		true
System.String	IdleTimeout		7200000
System.String	MaxConcurrentUsers		10
System.String	MaxShellRunTime		2147483647
System.String	MaxProcessesPerShell		25
System.String	MaxMemoryPerShellMB		1024
System.String	MaxShellsPerUser		30

Let us see the result of the following code:

```
Set-Item WSMAN:\wmf5node03\Shell\MaxShellsPerUser -Value 30
```

PS WSMAN:\wmf5node03\Shell> ls			
Type	Name	SourceOfValue	Value
System.String	AllowRemoteShellAccess		true
System.String	IdleTimeout		7200000
System.String	MaxConcurrentUsers		10
System.String	MaxShellRunTime		2147483647
System.String	MaxProcessesPerShell		25
System.String	MaxMemoryPerShellMB		1024
System.String	MaxShellsPerUser		5

Using the WSMAN cmdlets, we can manage the session's timeout period. Run the following command:

```
$Time = New-WSManSessionOption -operationtimeout 3000
Connect-WSMan -computer WMF5Node02 -sessionoption $Time
```

We can call `Disconnect-WSMan` to disconnect the client from the WinRM service. Once this is executed, we don't see the remote computer in the WSMAN drive.

Now, let's establish a connection with the remote LYNC 2013 server using the `WSManConnectionInfo` class using C# and PowerShell.

Following is a sample code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Management.Automation;
using System.Management.Automation.Remoting;
using System.Management.Automation.Runspaces;
using System.Security;
using System.Collections.ObjectModel;
using System.Text;

namespace Office365
{
    class Program
    {
        static void Main(string[] args)
        {
            string username = "DSCDEMOLAB\\SKBAdminID";
```

```

        string password = "SecureString";
        System.Security.SecureString securepassword = new
System.Security.SecureString();
        foreach (char c in password)
        {
            securepassword.AppendChar(c);
        }
        PSCredential credential = new
PSCredential(username, securepassword);
        WSManConnectionInfo connectioninfo = new
WSManConnectionInfo(new
Uri("https://RemoteServer/OcsPowershell"),
"http://schemas.microsoft.com/powershell/Microsoft.PowerShell",
credential);
        connectioninfo.AuthenticationMechanism =
AuthenticationMechanism.Default;
        connectioninfo.SkipCACheck = true;
        connectioninfo.SkipCNCheck = true;
        //connectioninfo.AuthenticationMechanism =
AuthenticationMechanism.Basic;
        connectioninfo.MaximumConnectionRedirectionCount =
2;
        //connectioninfo.MaximumConnectionRedirectionCount
= 2;
        using (Runspace runspace =
RunspaceFactory.CreateRunspace(connectioninfo))
        {
            runspace.Open();
            using (PowerShell powershell =
PowerShell.Create())
            {
                powershell.Runspace = runspace;
                //Create the command and add a parameter
                powershell.AddCommand("Get-CsUser");
                Collection<PSObject> results =
powershell.Invoke();
                foreach (PSObject result in results)
                {
                    Console.WriteLine(result.Properties["SamaccountName"].Value.ToString());
                    Console.ReadLine();
                }
            }
        }
    }
}

```

```
    }  
}
```

Refer to the following image:

```
string username = "DSCDEMOLAB\\SKBAdminID";  
string password = "SecureString"; ①  
System.Security.SecureString securepassword = new System.Security.SecureString(); ②  
foreach (char c in password)  
{  
    securepassword.AppendChar(c); ③  
} ④  
PSCredential credential = new PSCredential(username, securepassword); ⑤  
WSManConnectionInfo connectioninfo = new WSManConnectionInfo(new Uri("https://RemoteServer/OcsPowershell"), "http://schemas.microsoft.com/powershell/Microsoft.PowerShell", credential); ⑥  
connectioninfo.AuthenticationMechanism = AuthenticationMechanism.Default;  
connectioninfo.SkipCACheck = true; ⑦  
connectioninfo.SkipCNCheck = true;  
//connectioninfo.AuthenticationMechanism = AuthenticationMechanism.Basic;  
connectioninfo.MaximumConnectionRedirectionCount = 2;  
//connectioninfo.MaximumConnectionRedirectionCount = 2;
```

We used the `WSManConnectionInfo` class, which provides us with the connection information that is needed to connect to a remote runspace. Windows PowerShell uses a WinRM connection to connect to the computer on which the remote runspace is opened. Execute the following code:

```
WSManConnectionInfo connectioninfo = new  
WSManConnectionInfo(new  
Uri ("https://RemoteServer/OcsPowershell"),  
"http://schemas.microsoft.com/powershell/Microsoft.PowerShell",  
credential);  
"https://RemoteServer/OcsPowershell"
```

This is our remote Skype for Business server URL using which we can explore Skype for Business Web Front End Server (IIS). Using the preceding code, we can establish the connection and query information from the remote servers.

The same can be achieved using PowerShell. Execute the following code:

```
$cred = Get-Credential "Domain\SKBAdmin"  
$session = New-PSSession -ConnectionURI  
"https://RemoteServer/OcsPowershell" -Credential $cred  
Import-PSSession $session  
Error: Access denied due to incorrect credentials (401 status
```

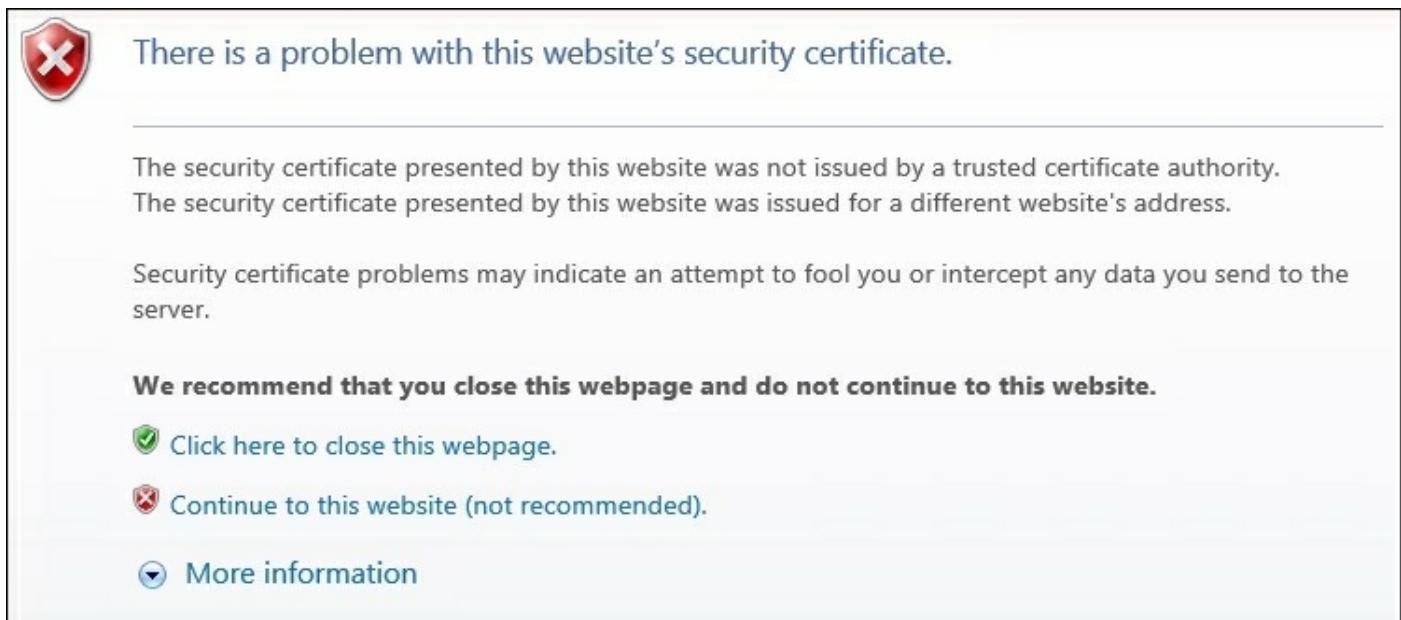
code)

The output of the code we just saw is shown in the following image:

```
PS C:\> $cred = Get-Credential "DSCDEMO\ChenV"
$session = New-PSSession -ConnectionUri "http://MFSNode02/OcsPowerShell" -Credential $cred
Import-PSSession $session
New-PSSession [Wfinode02] Connecting to remote server wfinode02 failed with the following error message : The WinRM client received an HTTP status code of 403 from the remote WS-Management service. For more information, see the about_Remote_Troubleshooting Help topic.
at Line:2 char:12
+ $session = New-PSSession -ConnectionUri "http://MFSNode02/OcsPowerSh ...
+ CategoryInfo          : OpenError: (System.Management.Automation.RemoteRunspace) [New-PSSession], PSRemotingTransportException
+ FullyQualifiedErrorId : 2144108273,PSSessionOpenFailed
Import-PSSession : Cannot validate argument on parameter 'Session'. The argument is null. Provide a valid value for the argument, and then try running the command again.
at Line:3 char:18
+ Import-PSSession $session
+ CategoryInfo          : InvalidData: () [Import-PSSession], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,Microsoft.PowerShell.Commands.ImportPSSessionCommand
```

For learning purpose, we can get the certificate information and add it to **Current User | Trusted Root Certification Authorities** and execute the PowerShell code—Yes! Connection establishes as expected.

To get the certificate information, you can go to the site, <https://RemoteServer/OCSPowerShell>. Take a look at the following image:



Click on the Lock button in your browser and get the information.

HTTP/HTTPS Listener

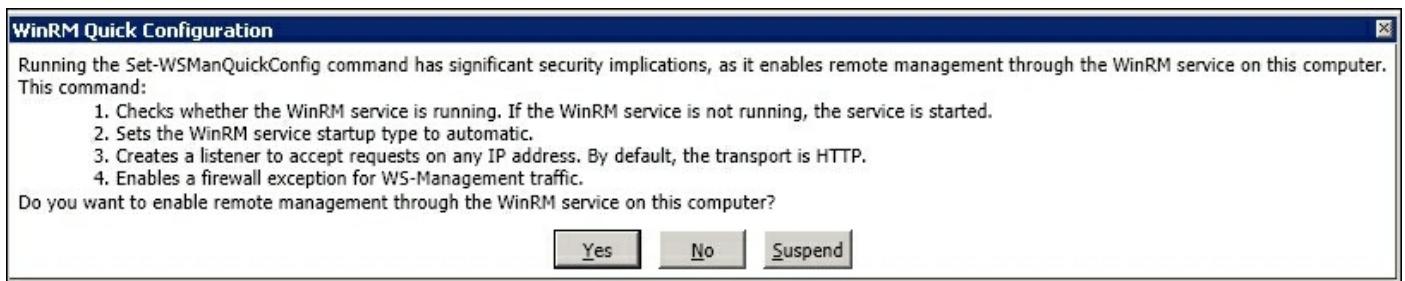
To configure the HTTP or HTTPS Listener, we need to ensure that the

ports, 5985 and 5986, are opened. The transport happens through the respective ports. Using PowerShell, we can easily enable the HTTP listener. Run the following command:

Set-WSManQuickConfig

```
PS C:\> Set-WSManQuickConfig
Set-WSManQuickConfig : Access is denied. You need to run this cmdlet from an elevated process.
At Line:1 char:1
+ Set-WSManQuickConfig
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Set-WSManQuickConfig], InvalidOperationException
+ FullyQualifiedErrorId : System.InvalidOperationException,Microsoft.WSMan.Management.SetWSManQuickConfigCommand
```

To avoid the access denied error, ensure that PowerShell is running as Administrator. This is applicable for Win 7 and 2008 server OS. The command works without elevated privileges if you run it under the local admin account. Take a look at the following image:



The output is as follows:

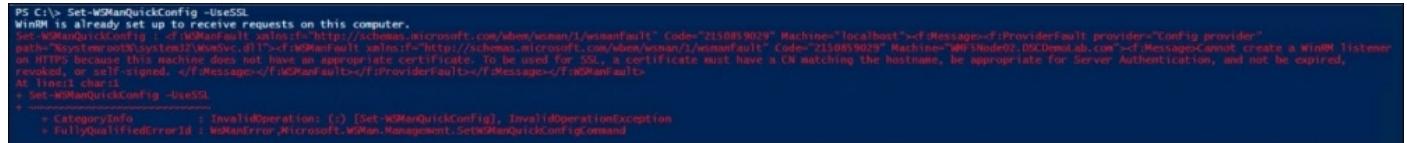
```
PS C:\Windows\system32> Winrm Enumerate Winrm/config/Listener
Listener
Address = *
Transport = HTTP
Port = 5985
Hostname
Enabled = true
URLPrefix = wsman
CertificateThumbprint
ListeningOn = <IPS>
```

By default, the HTTP listener will be created to enable HTTPS; we need to use the switch, -UseSSL.

Let's give it a try with the following command:

```
Set-WSManQuickConfig -UseSSL
```

The error is self-explanatory. This is the really cool stuff in Windows PowerShell. Take a look at the following image:



A screenshot of a Windows PowerShell window. The command entered is "Set-WSManQuickConfig -UseSSL". The output shows an error message: "WinRM is already set up to receive requests on this computer. To use SSL, the certificate at 'C:\Windows\system32\WinSvc.dll' must be valid and signed. A certificate must have a CN matching the hostname, be appropriate for Server Authentication, and not be expired, revoked, or self-signed. To be used for SSL, a certificate must have a CN matching the hostname, be appropriate for Server Authentication, and not be expired, revoked, or self-signed." The error code is 2150859029.

To overcome this, we can use the following code. You can check with your **certification authority (CA)** administrator for the certificate-related query:

```
$Cert = Get-ChildItem Cert:\LocalMachine\My | where {$_.Subject -match $env:COMPUTERNAME}  
"The installed SSL certificate: " + $Cert.Subject  
$CertPrivKey = $Cert.PrivateKey  
$KeyCertFile = Get-Item -path  
"$ENV:ProgramData\Microsoft\Crypto\RSA\MachineKeys\*" | where  
{$_.Name -eq  
$CertPrivKey.CspKeyContainerInfo.UniqueKeyContainerName}  
$KeyAcl = (Get-Item -Path  
$KeyCertFile.FullName).GetAccessControl("Access")  
$perm = "NT AUTHORITY\NETWORK SERVICE","Read","Allow"  
$accessRule = New-Object  
System.Security.AccessControl.FileSystemAccessRule $perm  
$KeyAcl.AddAccessRule($accessRule)  
Set-Acl $KeyCertFile.FullName $KeyAcl
```

Now, the following code will work:

```
Set-WSManQuickConfig -UseSSL
```

The output is as shown in the following code:

```
Listener  
Address = *  
Transport = HTTPS  
Port = 5986
```

```
Hostname  
Enabled = true  
URLPrefix = wsman  
CertificateThumbprint =  
D3438E6116227BA1459434943DB723C2C5D50C7C  
ListeningOn = <IPS>
```

In case this is a workgroup, we can add trusted hosts to accept the connections. This way, we can have more control in remoting. Your client accepts the connection only if it's listed in the trusted hosts.

Exploring CIM commands

We have covered a few basic examples of CIM and WMI in [Chapter 2, Unleashing Development Skills Using Windows PowerShell 5.0](#); let's take a quick review here and proceed with a few more examples using PowerShell and C#.

You may wonder, why do we need CIM, and why not WMI? In this topic, we'll discover how CIM helps IT professionals.

- **Common Information Model (CIM)** is the DMTF standard [DSP0004] for describing the structure and behavior of managed resources such as storage, network, or software components
- **Windows Management Instrumentation (WMI)** is a CIM server that implements the CIM standard on Windows
- The **WS-Management (WSMan)** protocol is a SOAP-based, firewall-friendly protocol for management clients to communicate with CIM servers
- **Windows Remote Management (WinRM)** is the Microsoft implementation of the WSMan protocol on Windows.

CIM provides a rich experience in PowerShell and supports standard compliance—yes; CIM does work in Windows- and nonWindows-based machines.

CIM is introduced in Windows PowerShell 3.0 with the 2012 server by default, but this is limited; it supports down-level OS as well.

To know all the available CIM cmdlets, use the following code:

```
Get-Command -Module CimCmdlets | Select Name
```

Following is a list of available CIM cmdlets:

- Export-BinaryMiLog
- Get-CimAssociatedInstance
- Get-CimClass
- Get-CimInstance
- Get-CimSession
- Import-BinaryMiLog
- Invoke-CimMethod
- New-CimInstance
- New-CimSession
- New-CimSessionOption
- Register-CimIndicationEvent
- Remove-CimInstance
- Remove-CimSession
- Set-CimInstance

Take a look at the following image:

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

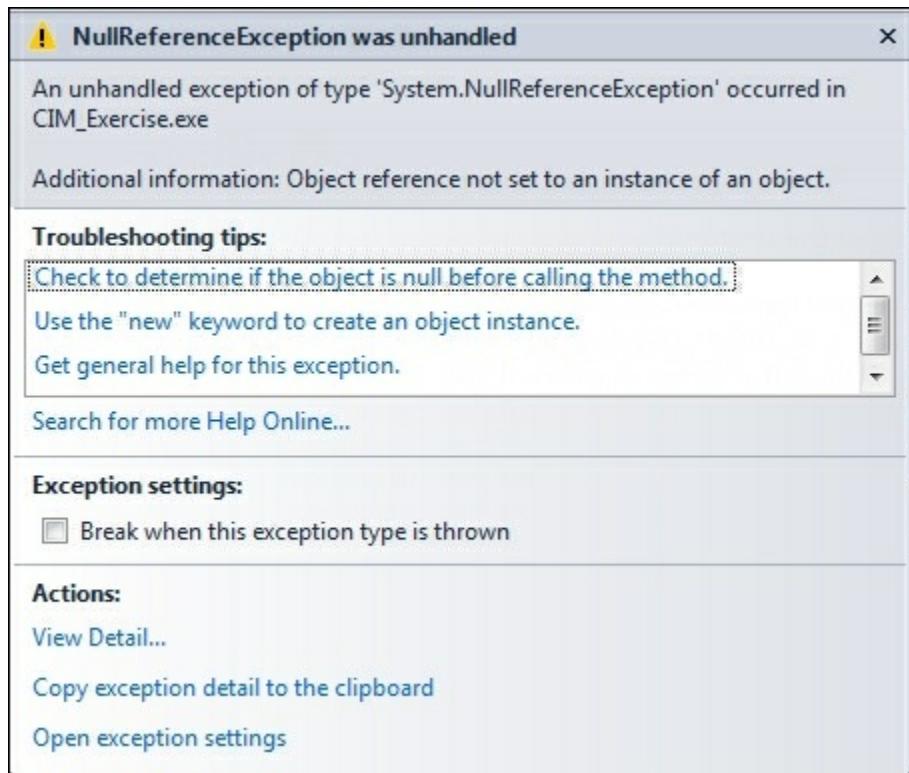
CIM cmdlets have auto tab completion, so it involves little typing and coding. Do remember that the default namespace is `root/cimv2`.

Now, let's take a look at the list of CIM classes:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Management.Automation;
using System.Management.Automation.Host;
using System.Collections.ObjectModel;

namespace CIM_Exercise
{
    class Program
    {
        static void Main(string[] args)
        {
            using (PowerShell PowerShellInstance =
PowerShell.Create())
            {
                PowerShellInstance.AddCommand("Get-CimClass");
                Collection<PSObject> result =
PowerShellInstance.Invoke();
                foreach (PSObject r in result)
                {
                    Console.WriteLine(r);
                    Console.ReadKey();
                }
            }
        }
    }
}
```

Ensure that you call a property that exists. If we don't use an existing property, the following error will appear:



After a successful execution, we will obtain the list as shown in the following image:

The screenshot shows a command-line window with the title bar 'file:///c:/Temp/CIM_Exercise/CIM_Exercise/bin/Debug/CIM_Exercise.EXE'. The window displays a list of CIM commands, each preceded by 'ROOT/CIMU2: __'. The commands listed are: SystemClass, thisNAMESPACE, Provider, Win32Provider, ProviderRegistration, EventProviderRegistration, ObjectProviderRegistration, ClassProviderRegistration, InstanceProviderRegistration, MethodProviderRegistration, PropertyProviderRegistration, EventConsumerProviderRegistration, NAMESPACE, IndicationRelated, EventFilter, EventConsumer, FilterToConsumerBinding, and AggregateEvent.

```
file:///c:/Temp/CIM_Exercise/CIM_Exercise/bin/Debug/CIM_Exercise.EXE
ROOT/CIMU2: __SystemClass
ROOT/CIMU2: __thisNAMESPACE
ROOT/CIMU2: __Provider
ROOT/CIMU2: __Win32Provider
ROOT/CIMU2: __ProviderRegistration
ROOT/CIMU2: __EventProviderRegistration
ROOT/CIMU2: __ObjectProviderRegistration
ROOT/CIMU2: __ClassProviderRegistration
ROOT/CIMU2: __InstanceProviderRegistration
ROOT/CIMU2: __MethodProviderRegistration
ROOT/CIMU2: __PropertyProviderRegistration
ROOT/CIMU2: __EventConsumerProviderRegistration
ROOT/CIMU2: __NAMESPACE
ROOT/CIMU2: __IndicationRelated
ROOT/CIMU2: __EventFilter
ROOT/CIMU2: __EventConsumer
ROOT/CIMU2: __FilterToConsumerBinding
ROOT/CIMU2: __AggregateEvent
```

Using help commands, we can explore all the CIM commands.

Exploring CIM methods

Using the `Invoke-CimMethod` cmdlet, we can perform operations such as creating, terminating, restarting, and so on.

In this exercise, let's consider the basic example of opening a notepad. Yes! We can simply type the word `notepad` in PowerShell, and this will do it. But we will explore the CIM methods using the `Invoke-CimMethod` cmdlet and its parameters.

The following is a single line of code in PowerShell that uses the `Invoke-CimMethod` cmdlet:

```
Invoke-CimMethod -ClassName Win32_Process -MethodName "Create"  
-Arguments @{"Commandline = "notepad.exe"}
```

The following is a single line of code in PowerShell that uses the `[WMICLASS]` type accelerator:

```
([wmiclass]"root\cimv2:Win32_Process") .Create('notepad.exe') ;
```

So, how do we find out the parameters of the methods? Run the following command and refer to the following image:

```
(Get-CimClass -ClassName Win32_Process) .CimClassMethods
```

PS C:\> (Get-CimClass -ClassName Win32_Process).CimClassMethods			
Name	ReturnType	Parameters	Qualifiers
Create	UInt32	{CommandLine, CurrentDirectory, ProcessStartupInformation, ProcessId}	{Constructor, Implemented, MappingStrings, Privileges...}
Terminate	UInt32	{Reason}	{Destructor, Implemented, MappingStrings, Privileges...}
GetOwner	UInt32	{Domain, User}	{Implemented, MappingStrings, ValueMap}
GetOwnersid	UInt32	{Sid}	{Implemented, MappingStrings, ValueMap}
SetPriority	UInt32	{Priority}	{Implemented, MappingStrings, ValueMap}
AttachDebugger	UInt32	{}	{Implemented, ValueMap}

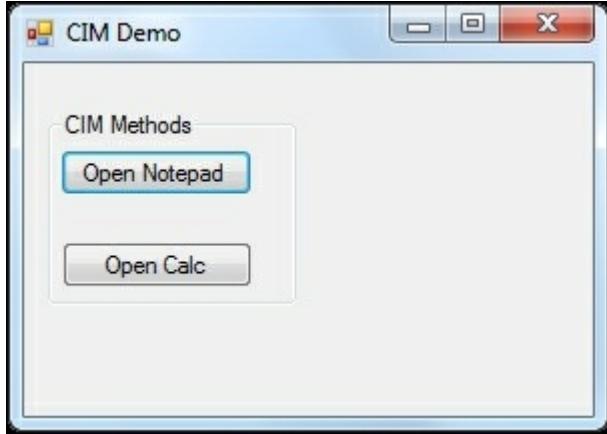
For each method name, we can see the parameters. It's easy to explore and use as required.

Now, let's create a small Windows form application and take note of the execution of CIM methods.

Following is the code used for this:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Management.Automation;
using System.Management.Automation.Runspaces;
namespace CIM_Exercise
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            using (PowerShell PowerShellInstance =
PowerShell.Create())
            {
                PowerShellInstance.AddScript("Invoke-CimMethod
-ClassName Win32_Process -MethodName Create -Arguments
@{CommandLine = notepad.exe}");
                PowerShellInstance.Invoke();
            }
        }
        private void button2_Click(object sender, EventArgs e)
        {
            using (PowerShell PowerShellInstance =
PowerShell.Create())
            {
                PowerShellInstance.AddScript("Invoke-CimMethod
-ClassName Win32_Process -MethodName Create -Arguments
@{CommandLine = calc.exe}");
                PowerShellInstance.Invoke();
            }
        }
    }
}
```

To avoid overuse of `Add` parameters in the PowerShell instance, we used the type accelerator method. The Windows form provides the options of **Open Notepad**, which opens the **Notepad** application, and **Open Calc**, which opens the **Calculator** application, as shown in the following screenshot:



Querying the remote machines using CIM

As we have previously seen, CIM supports down-level OS. You may wonder, can we not use CIM commands in servers running Windows 2008 with PowerShell 2.0? Not really, but we can establish a remote session and execute the CIM commands.

Consider a scenario where we need to query the OS-installed date on remote machines. WMI will do it, but we need to add a snippet of code for creating the time format, which is as follows:

```
(Get-WmiObject -Class Win32_operatingSystem -ComputerName "Remote") .InstallDate  
#20140326181309.000000+060  
(Get-CimInstance -ClassName CIM_OperatingSystem -ComputerName "Remote") .InstallDate  
#Wednesday, March 26, 2014 6:13:09 PM
```

Which one of these would our configuration management team prefer? Of course, they would prefer the one with the well-formatted date. Indeed, we can do this in WMI as well by adding a snippet of code as follows:

```
$os = (Get-WmiObject -Class Win32_operatingSystem)
[management.managementDateTimeConverter]::ToDateTime($os.Instal
lDate)
```

There are always multiple ways of doing things in PowerShell, and we can choose the one that best suits our requirement.

In this exercise, let's consider using the server remotely with PowerShell 2.0, and execute the CIM commands:

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

OS: 0.0.0 SP: 0.0 Stack: 2.0 proves that the remote machine is PowerShell 2.0.

So now, let's establish a session using the following code and get the OS-installed date:

```
$Dcom = New-CimSessionOption -Protocol Dcom
$session = New-CimSession -ComputerName RemoteServer -
SessionOption $Dcom
(Get-CimInstance -CimSession $session -ClassName
Win32_OperatingSystem).InstallDate
```

In this chapter, we explored the basics of Windows Remote Management; in the next chapter, we will discuss how to create DSC with MOF using class, deploying configuration, and the types of deployment modes.

Creating configuration scripts

In this topic, we will take a look at how to create a configuration using the `Configuration` and `Class` keywords. Configuration creates MOF, and this MOF needs to be pushed to or pulled by the target nodes. While using class, we will define the DSC resource and deploy the configurations.

Creating a configuration with MOF

We know that DSC is a declarative syntax, and this really helps developers and IT professionals to do more. In this section, we will see what developers can do with DSC.

We have discussed a few basic concepts of DSC and its stages; using this knowledge, let's build a web server. Most organizations face operational issues: the IT professional thinks that the infrastructure is good, and the code is at fault, but on the other hand, developers state that the code works fine in test but not in production. So, test and production are not identical. To fix this gap, we can use DSC and code our infrastructure.

Let's build a test web server using DSC. In this exercise, we will install IIS, as follows:

```
Configuration WebServer
{
    Node $env:COMPUTERNAME
    {
        WindowsFeature IIS
        {
            Name = 'Web-Server'
            Ensure = 'Present'
            IncludeAllSubFeature = $true
        }
    }
}
WebServer
```

While deploying this configuration, I did not note that my OS build version is 10074—Windows Server Technical Preview. This is a known issue, so the IIS installation failed as well, as shown in the following message:

```
Failed to start automatic updating for installed components.  
Error: 0x80040154
```

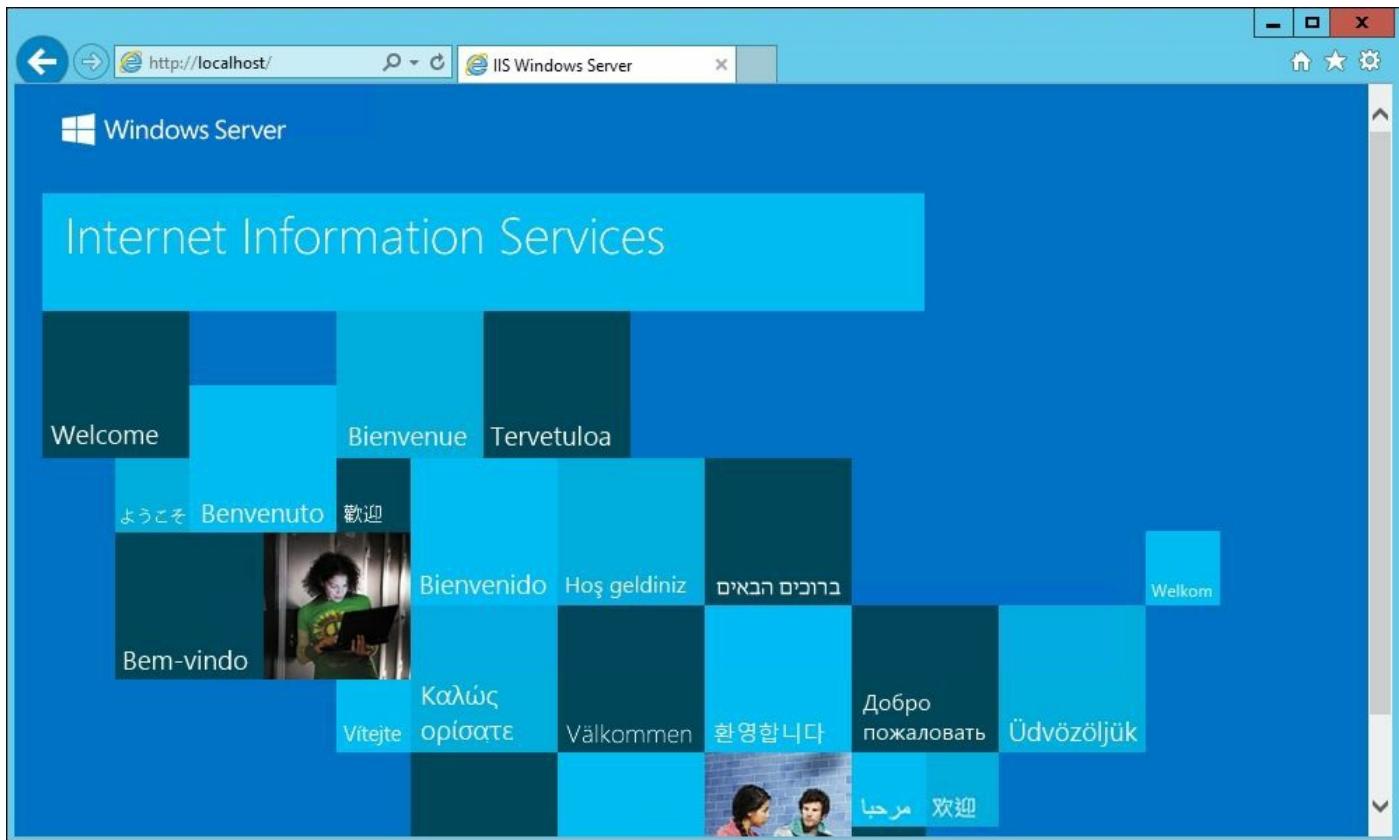
To fix this, we need to use `sconfig.cmd` and enter option 6—sounds weird! However, this is another bug in Technical Preview 2 Server OS, so let's download and install the updates.

This calls `Cscript.exe` to download and install updates; just give your system a reboot after this!

Now, we will fix the password issue and rerun the configuration script through the following code:

```
configuration WebServers  
{  
    Node $env:COMPUTERNAME  
    {  
        WindowsFeature WebServer  
        {  
            Name = 'Web-Server'  
            Ensure = 'Present'  
            IncludeAllSubFeature = $true  
        }  
    }  
}  
WebServers  
Start-DSCConfiguration .\WebServers -Wait -Verbose -Force
```

Yes, we can access localhost now, as shown in the following image:



Now, let's remove IIS—do remember that this requires a reboot. So, it's good to check the LCM Meta Configuration. All we need is action to continue after the reboot and reboot if needed, set to true. Execute the following code:

```
[DscLocalConfigurationManager()]
Configuration LCM
{
    Node localhost
    {
        Settings
        {
            RebootNodeIfNeeded = $true
            ActionAfterReboot = "ContinueConfiguration"
        }
    }
}
LCM
Set-DscLocalConfigurationManager .\LCM
```

To uninstall the Windows feature, we will use the same code; however,

we will change Present to Absent:

```
configuration WebServers
{
    Node $env:COMPUTERNAME
    {
        WindowsFeature WebServer
        {
            Name = 'Web-Server'
            Ensure = 'Absent'
            IncludeAllSubFeature = $true
        }
    }
}
```

The system reboots itself because we set the Meta. In the next topic, we will create a custom DSC resource using Class.

Creating a Class-defined DSC resource

To know more about Class, you can read the help document.

From version 5.0 onward, Windows PowerShell adds the language syntax to define classes and other user-defined types using formal syntax and semantics that are similar to other object-oriented programming languages.

You can refer to the release notes of WMF 5.0 for the class structure and more details.

Refer to the following link for the WMF 5.0 release notes on the skeleton class structure: <https://www.microsoft.com/en-us/download/details.aspx?id=46889>

Execute the following command:

```
enum Ensure
{
    Absent
    Present
}
[DscResource()]
```

```

class StringLiteral
{
    [DscProperty(Key)]
    [string]$Path

    [DscProperty(Mandatory)]
    [Ensure] $Ensure

    [DscProperty(Mandatory)]
    [string] $SourcePath
    [DscProperty(NotConfigurable)]
    [Nullable[datetime]] $CreationTime
    [void] Set()
    {
    }
    [bool] Test()
    {
    }

    [StringLiteral] Get()
    {
    }
}

```

Now, using the `Class` keyword, let's define a DSC Resource. The significant benefits of this are as follows:

- There is no need for schema MOF anymore
- A `DSCResource` folder inside the module folder is not needed
- Multiple DSC resources can be packaged as a single module script file

In this exercise, we will follow a step-by-step procedure to create a Class-based DSC resource, which will simply create a file or directory.

1. Refer to the TechNet article for reference using the URL
<https://technet.microsoft.com/en-us/library/dn948461.aspx>.
2. Create a folder structure. Run the following command:

```

$env: psmodulepath (folder)
  |- MyDscResources (folder)
    |- MyDscResource.psm1
      MyDscResource.psd1

```

3. Create a DSC resource using Class and save it as a PSM1 file.
4. Create a module manifest using the `New-ModuleManifest` cmdlet.
5. Test the configuration.

Let's create a DSC resource using the following code:

```
enum Ensure
{
    Absent
    Present
}

[DscResource()]
class MyTestClassResource{
    [DscProperty(Key)]
    [string]$Path

    [DscProperty(Mandatory)]
    [Ensure]$Ensure
    [DscProperty(Mandatory)]
    [ValidateSet("Directory","File")]
    [string]$ItemType
    #Replaces Get-TargetResource
    [MyTestClassResource] Get()
    {
        $Item = Test-Path $This.Path
        If($Item -eq $True)
        {
            $This.Ensure = [Ensure]::Present
        }
        Else
        {
            $This.Ensure = [Ensure]::Absent
        }
        Return $This
    }
    #Replaces Test-TargetResource
    [bool] Test()
    {
        $Item = Test-Path $This.Path
        If($This.Ensure -eq [Ensure]::Present)
        {
            Return $Item
        }
        Else
```

```

    {
        Return -not $Item
    }
}

#Replaces Set-TargetResource
[void] Set()
{
    $Item = Test-Path $This.Path
    If($This.Ensure -eq [Ensure]::Present)
    {
        If(-not $Item)
        {
            Write-Verbose "Creating Folder"
            New-Item -ItemType Directory -Path $This.Path
        }
    }
    #If [Ensure]::Absent
    Else
    {
        If($Item)
        {
            Write-Verbose "File exists and should be
absent. Deleting file"
            Remove-Item -Path $This.Path
        }
    }
}
}

```

Save the preceding code as a PSM1 file and create a module manifest.

Save the files in C:\Program

Files\WindowsPowerShell\Modules\ClassResourceDemo.

Save your class resource code as ClassResourceDemo.PSM1 in the folder that we created named ClassResourceDemo.

Now, create a module manifest using the New-ModuleManifest cmdlet, executing the following code:

```

New-ModuleManifest -Path
'C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Demo\DemoCl
ass.psdl'
-DscResourcesToExport 'MyTestClassResource' -PowerShellVersion
5.0 -Description 'Class based DSC resource' -ModuleVersion

```

```
'1.0.0.0' -Guid $([guid]::NewGuid()) -Author 'Chen' ` -RootModule '.\DemoClass.ps1' -CompanyName 'Something'
```

That's it! We are ready to test the resource using the following configuration code:

```
Configuration Test {
    Import-DscResource -ModuleName ClassResourceDemo
    Node Localhost {
        MyTestClassResource Testing
        {
            Ensure = 'Present'
            Path = 'C:\Test1'
            ItemType = 'Directory'
        }
    }
}
Test
```

Use the `Start-DscConfiguration` cmdlet to deploy the configuration, as shown in the following image:

```
PS C:\> Start-DscConfiguration .\Test -Wait -Verbose -Force
VERBOSE: Configuration .\Test is being applied...
VERBOSE: An LCM method call arrived from computer WMFSNODE01 with user sid S-1-5-21-3437098744-4265828018-247196969-500.
VERBOSE: [WMFSNODE01]: LCM: [ Start Set
VERBOSE: [WMFSNODE01]: LCM: [ Start Resource ] [MyTestClassResource]Testing]
VERBOSE: [WMFSNODE01]: LCM: [ Start Test ] [MyTestClassResource]Testing
VERBOSE: [WMFSNODE01]: LCM: [ End Test ] [MyTestClassResource]Testing in 0.0150 seconds.
VERBOSE: [WMFSNODE01]: LCM: [ Start Set ] [MyTestClassResource]Testing
VERBOSE: [WMFSNODE01]: LCM: [ Start Resource ] [MyTestClassResource]Creating Folder
VERBOSE: [WMFSNODE01]: LCM: [ End Set ] [MyTestClassResource]Testing in 0.0320 seconds.
VERBOSE: [WMFSNODE01]: LCM: [ End Set Resource ]
VERBOSE: [WMFSNODE01]: LCM: [ End Set ] in 0.2500 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 0.3 seconds.
```

How does this work?

Take a look at the eight main steps in the following image:

```

enum Ensure ①
{...}

[DscResource()] ② ③
class MyTestClassResource{}

    [DscProperty(Key)] ④
    [string]$Path

    [DscProperty(Mandatory)] ⑤
    [Ensure]$Ensure

    [DscProperty(Mandatory)]
    [ValidateSet("Directory", "File")]
    [string]$ItemType

    #Replaces Get-TargetResource
    [MyTestClassResource] Get() ⑥
    {...}

    #Replaces Test-TargetResource
    [bool] Test() ⑦
    {...}

    #Replaces Set-TargetResource
    [void] Set() ⑧
    {...}

}

```

- **1:** Here, we used the `enum` keyword and named it `Ensure`
- **2:** Here, we used the `[DscResource()]` attribute to specify that the class is a DSC resource
- **3:** Here, we created a class named `MyTestClassResource`
- **4:** Here, we declared DSC properties—this value will be set before calling the instance
- **5:** The `Ensure` property is the enum type and is set to `Mandatory`
- **6:** The `Get()` method returns the instance of the class
- **7:** The `Test()` method returns a boolean value, `true` or `false`; this is called first, so check whether or not the resource is in the desired state
- **8:** The `Set()` method sets the resources to the desired state

Note

Here are a few key things to note:

- Inside `enum`, no numbers are allowed

- No separators are required in `enum`
- While creating a module manifest, do remember to use the `DscResourcesToExport` parameter and pass your classname; or else, you won't see the resource in the `Get-DscResource` output.
- You can create multiple class files and make use of the resources

Now that you have created your first Class-based DSC resource, it's easy and in a human, very readable format. To know more about Class and writing custom DSC resources, use the help document as follows:

```
help New-ModuleManifest -Detailed  
help about_Classes -Detailed  
help about_Modules -Detailed
```

Types of deployment modes

Windows PowerShell has two configuration modes: push and pull. In this topic, we will learn about implementing both push and pull modes. Before we start discussing the types of configuration modes, let's recap the high-level details of DSC.

Using Windows PowerShell, we will write a configuration. This contains the elements we need to configure on target nodes.

After the successful execution of the configuration script, we get a Managed Object Format file. The MOF files are then distributed to the target nodes.

Local Configuration Manager in the target node will parse the received configurations and ensure that the nodes are in the desired state.

Distributing the MOF files to the target node is very important. To carry this out, we need to follow either a push or pull method. Most organizations use the pull mode for the reasons of standard, security, and reliability.

The push mode

As the term implies, the push mode is unidirectional, and the action is immediate. The configurations are pushed to the target nodes and set to be in the desired state. Using the `Start-DscConfiguration` command, the configurations are applied to target nodes.

In this exercise, let's push the configuration to the remote nodes. Execute the following code:

```
Configuration PushDemo
{
    param([Parameter(Mandatory = $true)]$nodeName)
    Node $nodeName
    {
        Service Bits
```

```

    {
        Name = "Bits"
        Ensure = "Present"
        State = "Running"
    }
}

PushDemo -NodeName WMF5Node02
Start-DscConfiguration .\PushDemo -ComputerName WMF5Node02 -
Verbose -Wait -Force

```

This is how the configurations are pushed to the target nodes. The output is illustrated in the following image:

```

PS C:\> Start-DscConfiguration .\PushDemo -ComputerName WMF5Node02 -Verbose -Wait -Force
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, {'methodName' = SendConfigurationApply,'className' = MSFT_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer WMF5NODE03 with user sid S-1-5-21-3437098744-4265828018-247196969-500.
VERBOSE: [WMF5NODE02]: LCM: [ Start Set      ]
VERBOSE: [WMF5NODE02]: LCM: [ Start Resource ] {[Service]Bits}
VERBOSE: [WMF5NODE02]: LCM: [ Start Test     ] {[Service]Bits}
VERBOSE: [WMF5NODE02]: LCM: [ End  Test     ] {[Service]Bits} in 0.2190 seconds.
VERBOSE: [WMF5NODE02]: LCM: [ Skip Set      ] {[Service]Bits}
VERBOSE: [WMF5NODE02]: LCM: [ End  Resource ] {[Service]Bits}
VERBOSE: [WMF5NODE02]: LCM: [ End  Set      ] {[Service]Bits}
VERBOSE: [WMF5NODE02]: LCM: [ End  Set      ] in 1.2030 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 1.321 seconds

```

The pull mode

In the DSC pull mode, a few additional steps are needed. As in the push mode, we can't easily carry out the steps, but we can configure the pull server with a few steps.

In the pull mode, the target nodes are the pull client and where we have the actual configurations is called the pull server. The basic process involves creating a Desired State Configuration for the target node and setting up a pull server and DSC on the target node.

In this topic, we will discuss building a pull server using the SMB share and HTTP/HTTPS protocols.

Creating a pull server using the SMB share

Perform the following steps:

1. Create a network share in your pull server.
2. Configure your pull client, which is the Local Configuration

Manager (LCM) engine.

3. Create a configuration.
4. Use the pull client to pull the configuration from the pull server.

We will create the SMB share in the WMF5Node01—domain controller. Execute the following code:

```
#Creates a Directory
New-Item C:\ -Name DSCShare -ItemType Directory -Verbose
#Creates a SMB Share and gives read access to everyone
New-SmbShare -Name SMBPullServer -Path C:\DSCShare -ReadAccess
Everyone -Description "PULL server SMB" -verbose
```

That's it! There's no more work in server for us. Let's log in to the client nodes, or connect to WMF5Node03, to configure LCM. Execute the following code:

```
Configuration SMBClient {
Node Localhost {
LocalConfigurationManager {
ConfigurationID = 'e86549dc-7a5f-45b6-9d66-560d980587a8'
RefreshMode = "Pull"
DownloadManagerName = "DscFileDownloadManager"
DownloadManagerCustomData = @{
SourcePath =
"\WMF5Node01\DSCShare"
}
ConfigurationModeFrequencyMins = 30
RefreshFrequencyMins = 20
}
}
}
SMBClient
Set-DscLocalConfigurationManager -Path .\SMBClient
```

What we should see after the execution of the preceding code is the output of the `Get-DscLocalConfigurationManager` cmdlet—observe the `RefreshMode`—changed to `Pull`. Take a look at the following image:

```
PS C:\> Get-DscLocalConfigurationManager

ActionAfterReboot          : ContinueConfiguration
AgentId                    : 4FA44C3A-3206-11E5-80C0-000D3A2135E3
AllowModuleOverWrite       : False
CertificateID              :
ConfigurationDownloadManagers : {}
ConfigurationID            : c720b906-5e47-4ff1-b34c-24bed9905770
ConfigurationMode          : ApplyAndMonitor
ConfigurationModeFrequencyMins : 50
Credential                 :
DebugMode                  : {NONE}
DownloadManagerCustomData : {MSFT_KeyValuePair (key = "SourcePath")}
DownloadManagerName        : DscFileDownloadManager
LCMCompatibleVersions      : {1.0, 2.0}
LCMState                   : Idle
LCMStateDetail             :
LCMVersion                 : 2.0
MaxPendingConfigRetryCount : 10
StatusRetentionTimeInDays  :
PartialConfigurations     :
RebootNodeIfNeeded         : False
RefreshFrequencyMins       : 30
RefreshMode                : Pull
ReportManagers             : {}
ResourceModuleManagers     : {}
PSCoputerName              :
```

Pull Mode

Now, it's time for us to create the DSC resources and store them in the pull server. Execute the following code:

```
Configuration SMBDemoConfig {
    Node 'c720b906-5e47-4ff1-b34c-24bed9905770' {
        Service Bits {
            Name = 'Bits'
            State = 'Running'
        }
    }
}

SMBDemoConfig -OutputPath C:\DSCShare
```

After this, we will create a checksum file in the same shared location through the following code:

```
New-DscChecksum -ConfigurationPath C:\DSCShare -OutPath
C:\DSCShare -Force
```

Now, at the C:\DSCShare location, we will see the following two files:

- The c720b906-5e47-4ff1-b34c-24bed9905770.mof file
- The c720b906-5e47-4ff1-b34c-24bed9905770.mof.checksum file

The reason we created the checksum file is that this is used to compute the configuration difference between the local configuration store and the one which is available in the actual pull server.

That's it! We are done. Now, either we can wait for the configuration to be applied or simply execute the following code:

```
Start-ScheduledTask -TaskName "Consistency" -TaskPath  
"\Microsoft\Windows\Desired State Configuration\"
```

Creating the pull server using HTTP and HTTPS

Using HTTP and HTTPS is an optimal way for the IT infrastructure.

The pull server has two HTTP web services that manage the following:

- The configuration(s) hosted on that server
 - DSC compliance

DSC compliance is used for a periodical checking of the configuration nodes synchronization.

These services each listen on a turned off TCP port. We will now configure 8080 (pull server) and 9080 (compliance). Execute the following code:

```


EndpointName          = "PSDSCPullServer"
Port                  = 8080
PhysicalPath          =
"$env:SystemDrive\inetpub\wwwroot\PSDSCPullServer"
    CertificateThumbPrint = "AllowUnencryptedTraffic"
    ModulePath           =
"$env:PROGRAMFILES\WindowsPowerShell\DscService\Modules"
    ConfigurationPath    =
"$env:PROGRAMFILES\WindowsPowerShell\DscService\Configuration"
    State                = "Started"
    DependsOn            =
[WindowsFeature]DSCServiceFeature"
}
xDscWebService PSDSCComplianceServer
{
    Ensure          = "Present"
    EndpointName   = "PSDSCComplianceServer"
    Port           = 9080
    PhysicalPath   =
"$env:SystemDrive\inetpub\wwwroot\PSDSCComplianceServer"
    CertificateThumbPrint = "AllowUnencryptedTraffic"
    State          = "Started"
    IsComplianceServer = $true
    DependsOn      = (
[WindowsFeature]DSCServiceFeature",
[xDSCWebService]PSDSCPullServer")
}
}
ConfigurePullServer -ComputerName localhost


```

The preceding code performs three simple tasks:

- It installs the DSC service on the localhost
- It generates a pull server web service—port 8080
- It generates a compliance server web service—port 9080

Note that we didn't install IIS and tools in the server. So, we won't explore it in this exercise.

We need to explore C:\Install backup folder, that contains the MOF file we need.

Carry out the same steps as you did with the SMB pull server

configuration. Set up a pull client, as shown in the following code:

```
Configuration SetPullMode
{
    # param([string]$guid)
    Node WMF5Node03.DSCDemoLab.Com
    {
        LocalConfigurationManager
        {
            ConfigurationMode = "ApplyAndAutoCorrect"
            ConfigurationModeFrequencyMins = 15
            ConfigurationID = "479316c3-712a-4e5a-9b87-
4fde1bf0433e"
            RefreshMode = 'Pull'
            DownloadManagerName = 'WebDownloadManager'
            DownloadManagerCustomData = @{
                ServerUrl =
'http://WMF5Node01.DSCDemoLab.Com:8080/PSDSCPullServer.svc';
                AllowUnsecureConnection = 'true' }
        }
    }
}

SetPullMode
Set-DSCLocalConfigurationManager -Computer
WMF5Node03.DSCDemoLab.Com -Path "C:\SetPullMode" -Verbose
```

Then start creating a Configuration.

```
Configuration InstallBackup {
    param (
        [string[]]$ComputerName = "localhost"
    )

    Node $ComputerName {
        WindowsFeature Backup {
            Ensure = "Present"
            Name   = "Windows-Server-Backup"
        }
    }
}
InstallBackup -ComputerName WMF5Node03.DSCDemolab.com
```

It is important to enact the configuration we need to move the MOF and

checksum MOF files to C:\Program
Files\WindowsPowerShell\DscService\Configuration.

We only need to create a checksum file, and we are done!

HTTPS is a more secure way, so let's quickly take a look at the additional steps required to configure the HTTPS pull server:

1. Request and install a Web Server SSL certificate on the DSC pull server from a trusted certificate authority.
2. Get the certificate thumbprint. It will be needed for the pull server configuration. The best way to accomplish this task is to use PowerShell and the CERT: drive. So, execute the following code:

```
Get-ChildItem -Path cert: -Recurse |  
Where { $_.FriendlyName -like "Web Server Ceritificate" } |  
select Subject, FriendlyName, Thumbprint | Format-List
```

3. Now, in the B04702_03_04.ps1 code file, change the line number 21 to the following along with the certificate thumbprint we generated and follow the same procedure as with the HTTP pull server configuration:

```
CertificateThumbPrint = "AllowUnencryptedTraffic"
```

Summary

In this chapter, we explored the basic concepts of Desired State Configuration, such as installing WMF 5.0 April 2015 Preview and imperative versus declarative programming and their significant features. If you think that built-in resources are limited, start building your own DSC resources and sharing them with the community. Using DSC, we can manage on-premises, and we can also manage the cloud environment using a declarative syntax. If you wish to learn about building Azure VM using DSC, refer to the following link:

<http://blogs.msdn.com/b/powershell/archive/2014/08/07/introducing-the-azure-powershell-dsc-desired-state-configuration-extension.aspx>

In the next chapter, we will cover PowerShell and web technologies, in which we will have exercises based on JSON, REST API, PowerShell Web Access, Web services, OData extensions, and PowerShell Web Access. The examples in the next chapter will cover managing Office 365 environments using PowerShell and C#.

Chapter 4. PowerShell and Web Technologies

IT professionals and developers require Windows PowerShell for various reasons, such as the following:

- To perform administration tasks
- For a continuous delivery
- To manage/automate deployments

The use of PowerShell is increasing day by day, which helps us to experience the growth and significant improvements of Windows PowerShell. IT professionals and developers can remotely operate servers from their mobiles and perform tasks on the fly.

In this chapter, we will cover the following:

- **PowerShell Web Access**
- **OData IIS Extension**
- Exploring web requests and web services
- Exploring the REST API and JSON

PowerShell Web Access

PowerShell Web Access (PSWA) was introduced in Windows Server 2012. In short, PSWA acts as a gateway to run the PowerShell cmdlets from a remote computer. The key benefit of this is that we don't need any remote management software on remote computers.

PSWA can be implemented using the following three steps:

1. Installing the PowerShell Web Access gateway
2. Configuring the gateway
3. Configuring authorization rule

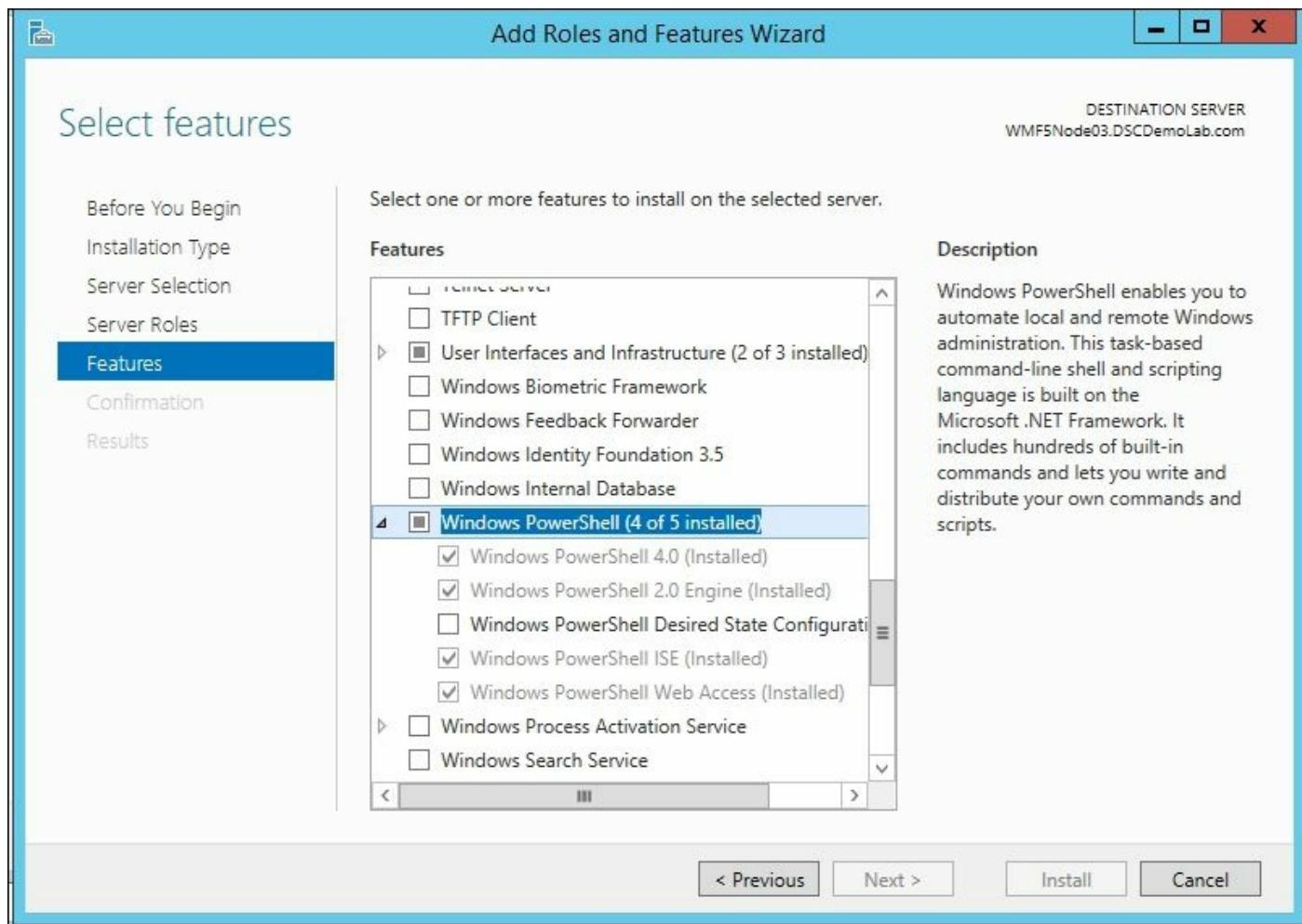
Installing PowerShell Web Access

We can do a PSWA installation using GUI and PowerShell. Let's explore these methods one by one.

Installing Windows PowerShell Web Access using GUI is a straightforward approach. Yes! It's just click—click, and you're done. Perform the following steps:

1. Log in to the Windows 2012 server.
2. Click on Start icon.
3. Select **Control Panel**.
4. Alternatively, you can simply click on **Server manager** if you are familiar with Windows Server.
5. Now, we will see **Add Roles and Features Wizard**.
6. Windows PowerShell Web Access can be found under the **Features** tab; keep clicking on **Next** and you'll find it.
7. Select Windows PowerShell Web Access.
8. Click on **Install**.

After installation, the **Features** tab looks similar to the following image:



Using Windows PowerShell, we can install the Windows PowerShell Web Access feature through the following command:

```
Install-PswaWebApplication -UseTestCertificate -Verbose
```

Note that we haven't configured PowerShell Web Access yet; we have only installed it.

After the execution of the preceding PowerShell code, we can see PSWA on our server:

<https://localhost/pswa>

?

Windows PowerShell Web Access

Enter your credentials and connection settings

User name:	<input type="text"/>
Password:	<input type="password"/>
Connection type:	Computer Name <input type="button" value="▼"/>
Computer name:	<input type="text"/>
<input checked="" type="checkbox"/> Optional connection settings	
<input type="button" value="Sign In"/>	

© 2013 Microsoft Corporation. All rights reserved.

In the preceding example, we used the `UseTestCertificate` property. However, this is not recommended for the production environment, so you need to check with the CA admin and get a valid certificate.

Now, let's take a look at the cmdlets available in the PowerShell Web Access module:

```
Get-Command -Module PowerShellWebAccess
```

CommandType	Name	Version	Source
Function	Install-PswaWebApplication	1.1.0.0	PowerShellWebAccess
Function	Uninstall-PswaWebApplication	1.1.0.0	PowerShellWebAccess
Cmdlet	Add-PswaAuthorizationRule	1.1.0.0	PowerShellWebAccess
Cmdlet	Get-PswaAuthorizationRule	1.1.0.0	PowerShellWebAccess
Cmdlet	Remove-PswaAuthorizationRule	1.1.0.0	PowerShellWebAccess
Cmdlet	Test-PswaAuthorizationRule	1.1.0.0	PowerShellWebAccess

Now let's take a look at the procedure to configure PowerShell Web Access with a few, custom settings. Let's uninstall PowerShell Web Access using the following PowerShell cmdlet:

```
Uninstall-PswaWebApplication -WebApplicationName pswa -Verbose  
-DeleteTestCertificate
```

The output is illustrated in the following image:

```
PS C:\> Uninstall-PswaWebApplication -WebApplicationName pswa -Verbose -DeleteTestCertificate  
Removing web application pswa...  
  
Removing application pool pswa_pool...  
  
Removing self-signed certificate...  
PS C:\>
```

This removes the following:

- The PSWA web application
- The pswa_pool application pool
- The self-signed certificate

This will retain the Windows PowerShell feature.

Let's configure PSWA using Desired State Configuration. Execute the following code:

```
Configuration PSWA  
{  
    WindowsFeature IIS {  
        Name = 'Web-Server'  
        Ensure = 'Present'  
        IncludeAllSubFeature = $true  
    }  
    WindowsFeature PS {  
        Name = "WindowsPowerShellWebAccess"  
        Ensure = "Present"  
        IncludeAllSubFeature = $true  
    }  
    Script PSWA {  
        SetScript = {  
            Install-PswaWebApplication  
        }  
        TestScript = {  
            [bool] (Get-WebApplication -Name pswa | where
```

```

{$_._ApplicationPool -eq 'pswa_pool'}) -eq $true
}
GetScript = {
    $result = [bool](Get-WebApplication -Name pswa |
where {$_._ApplicationPool -eq 'pswa_pool'})
    return @{Installed = $result}
}
}
PSWA

```

In the preceding code, we used the Configuration keyword and installed the Web Server feature and Windows PowerShell Web Access feature. Then, using the script resource, we installed PSWA.

In the script resource, the test script is executed first to test whether the Web application and pool exists for the default pswa and pswa_pool. The configuration happens dynamically.

Configuring PowerShell Web Access

Why do we need to do this? It's necessary to meet organization standards, naming conventions, and branding. Along with this, we need to use a valid certificate in the production environment. Remember that using a test certificate is not recommended practice for the production environment. However, as we will do this in a lab, we will use – UseTestCertificate.

Using the following steps, we can insert organization name in the login page:

1. Open IIS.
2. Explore the **Default Web Site**.
3. Right-click on **pswa** and select **Explore**.
4. The default location is
C:\Windows\Web\PowerShellWebAccess\wwwroot.
5. Copy the `FormLayout.Master` file to the desired location and modify the code as required.

To change the organization name, just add the following piece of code:

```
this.document.title = "We are Testing this!" +  
L_FeatureName_Text;
```

Simply replace the document title text with your own text and load the page: <https://localhost/pswa>.

Applying authorization rules

We have installed and customized PSWA. Now, to use it, we need to apply authorization rules.

To do this, we will use the `Add-PswaAuthorizationRule` cmdlet, as follows:

```
help Add-PswaAuthorizationRule -Examples
```

Only for the purposes of the lab, we run the following command:

```
Add-PswaAuthorizationRule -UserName * -ComputerName * -  
ConfigurationName *
```

Remember that this is to turn off authorization rules and simply allow all the users to access all the endpoints in the computer. This is, however, not good for production. You can plan this for groups and delegate the access as required.

Using the following code, we will provide access to one admin ID:

```
$param = @{  
    UserName = 'DSCDEMOLAB\ChenV'  
    ComputerName = $env:COMPUTERNAME  
    ConfigurationName = 'Microsoft.PowerShell'  
    Force = $true  
    Verbose = $true  
}  
Add-PswaAuthorizationRule @param
```

We used the `Force` parameter to avoid the following prompt:



Once the preceding code is executed, we will get the following output:

Id	RuleName	User	Destination	ConfigurationName
0	Rule 0	DSCDEMOLAB\ChenV	WMF5Node03.DSCDemoLab.com	Microsoft.PowerShell

Now, we can use this application in our custom applications based on the scenario and requirements. All we need to ensure is security.

Management OData IIS Extensions

OData, which stands for **Open Data Protocol**, is a RESTful web protocol. OData is based on HTTP and JSON, and using this, we can perform operations such as querying and updating.

We will cover the `ODataUtils` class before starting with OData IIS Extensions.

OData allows us to create a client-side module based on the OData endpoints. This is based on CDXML. Just take a look at the `Microsoft.PowerShell.ODataUtils` module.

This module has a single command, `Export-ODataEndpointProxy`, but is very powerful for exploring management data.

Let's create a client-side module using the following code:

```
$Odata = @{
    Uri =
    'http://services.odata.org/v3/(S(snyobsk1hhutkb2yulwldgf1))/odata/odata.svc'
    MetadataUri =
    'http://services.odata.org/v3/(S(snyobsk1hhutkb2yulwldgf1))/odata/odata.svc/$metadata'
    OutputModule = 'C:\Temp\DemoModule'
    AllowUnSecureConnection = $true
}
Export-oDataEndpointProxy @Odata
```

The output is illustrated in the following image:

Directory: C:\Temp\DemoModule

Mode	LastWriteTime	Length	Name
-a---	7/30/2015 1:40 PM	3426	ComplexTypeDefinitions.psm1
-a---	7/30/2015 1:40 PM	14469	Product.cdxml
-a---	7/30/2015 1:40 PM	8094	ProductDetail.cdxml
-a---	7/30/2015 1:40 PM	7978	Category.cdxml
-a---	7/30/2015 1:40 PM	9304	Supplier.cdxml
-a---	7/30/2015 1:40 PM	8042	Person.cdxml
-a---	7/30/2015 1:40 PM	9832	PersonDetail.cdxml
-a---	7/30/2015 1:40 PM	8506	Advertisement.cdxml
-a---	7/30/2015 1:40 PM	714	ServiceActions.cdxml
-a---	7/30/2015 1:40 PM	7470	DemoModule.psd1



Module

In the preceding example, we have used the following parameters:

- The `Uri` parameter
- The `MetadataUri` parameter
- The `OutputModule` parameter
- The `AllowUnsecureConnection` parameter

Use the `Import-Module` cmdlet to use this module. Take a look at the following image:

```
PS C:\> Import-Module C:\Temp\DemoModule -Verbose
VERBOSE: Loading module from path 'C:\Temp\DemoModule\DemoModule.psd1'.
VERBOSE: Loading module from path 'C:\Temp\DemoModule\ComplexTypeDefinitions.psm1'.
VERBOSE: Loading module from path 'C:\Temp\DemoModule\ServiceActions.cdxml'.
VERBOSE: Loading module from path 'C:\Temp\DemoModule\Product.cdxml'.
VERBOSE: Importing function 'Get-Product'.
VERBOSE: Importing function 'New-Product'.
VERBOSE: Importing function 'Remove-Product'.
VERBOSE: Importing function 'Set-Product'.
VERBOSE: Loading module from path 'C:\Temp\DemoModule\ProductDetail.cdxml'.
VERBOSE: Importing function 'Get-ProductDetail'.
VERBOSE: Importing function 'New-ProductDetail'.
VERBOSE: Importing function 'Remove-ProductDetail'.
VERBOSE: Importing function 'Set-ProductDetail'.
VERBOSE: Loading module from path 'C:\Temp\DemoModule\Category.cdxml'.
VERBOSE: Importing function 'Get-Category'.
VERBOSE: Importing function 'New-Category'.
VERBOSE: Importing function 'Remove-Category'.
VERBOSE: Importing function 'Set-Category'.
VERBOSE: Loading module from path 'C:\Temp\DemoModule\Supplier.cdxml'.
VERBOSE: Importing function 'Get-Supplier'.
VERBOSE: Importing function 'New-Supplier'.
VERBOSE: Importing function 'Remove-Supplier'.
VERBOSE: Importing function 'Set-Supplier'.
VERBOSE: Loading module from path 'C:\Temp\DemoModule\Person.cdxml'.
VERBOSE: Importing function 'Get-Person'.
```



Import-Module - Result

To explore the available commands, we can use the following cmdlet:

`Get-Command -Module DemoModule`

CommandType	Name	Version	Source
Function	Get-Advertisement	1.0	DemoModule
Function	Get-Category	1.0	DemoModule
Function	Get-Person	1.0	DemoModule
Function	Get-PersonDetail	1.0	DemoModule
Function	Get-Product	1.0	DemoModule
Function	Get-ProductDetail	1.0	DemoModule
Function	Get-Supplier	1.0	DemoModule
Function	New-Advertisement	1.0	DemoModule
Function	New-Category	1.0	DemoModule
Function	New-Person	1.0	DemoModule
Function	New-PersonDetail	1.0	DemoModule
Function	New-Product	1.0	DemoModule
Function	New-ProductDetail	1.0	DemoModule
Function	New-Supplier	1.0	DemoModule
Function	Remove-Advertisement	1.0	DemoModule
Function	Remove-Category	1.0	DemoModule
Function	Remove-Person	1.0	DemoModule
Function	Remove-PersonDetail	1.0	DemoModule
Function	Remove-Product	1.0	DemoModule
Function	Remove-ProductDetail	1.0	DemoModule
Function	Remove-Supplier	1.0	DemoModule
Function	Set-Advertisement	1.0	DemoModule
Function	Set-Category	1.0	DemoModule
Function	Set-Person	1.0	DemoModule
Function	Set-PersonDetail	1.0	DemoModule
Function	Set-Product	1.0	DemoModule
Function	Set-ProductDetail	1.0	DemoModule
Function	Set-Supplier	1.0	DemoModule

Available Commands

Now, we can use the following commands to get the management data information:

PS C:\> Get-Person -AllowUnsecureConnection	
ID	Name
0	Paula Wilson
1	Jose Pavarotti
2	Art Braunschweiger
3	Liz Nixon
4	Liu Wong
5	Jaime Yorres
6	Fran Wilson

Get-Person [cmdlet] output

We can also modify this module to suit our requirements. The OData endpoint we used is a read-only type, so we can't set the values for now.

Note

You can open and explore CDXML. Don't break this because it's not

easy to fix.

Creating the Management OData web service

The Management OData IIS extension is an infrastructure using which we can create an ASP.NET web service endpoint. This web service contains management data, which can be accessed using Windows PowerShell. In this exercise, let's create an OData web service; for this, we need the following:

- **OData Schema Designer**
- Features such as `IIS-WebServerRole`, `IIS-WebServer`, `IIS-HttpTracing`, and `IIS-Management OData`

Use the following link to download Management OData Schema Designer:

<https://visualstudiogallery.msdn.microsoft.com/77bb35c1-7695-4f5e-ba1a-9ffeb6fe0d14>

There are handy codes available in gallery to create Management OData Web Service; they can be found at the following site:

<https://code.msdn.microsoft.com/windowsdesktop/PswsRoleBasedPlugIn9c79b75a>

By following these steps, we can create Management OData Web Service.

Exploring web requests

Web requests help us request data from **Uniform Resource Identifier (URI)**. A developer performs many tasks using a web request, such as finding a response, downloading files, and reading files from the Internet.

In this topic, we will take a look at the cmdlets used for web requests and the `WebRequest` class.

Using the Windows PowerShell cmdlet, `Invoke-WebRequest`, we can perform a few tasks. Run the following command:

```
#Read the help document  
help Invoke-WebRequest -Detailed
```

This gives the help content.

```
#Using the command  
$site = Invoke-WebRequest http://www.Bing.com  
$site | GM
```

This lists the members (Methods and Properties)

```
#Checking the Status Code  
"The Site Status Code is `t" + $site.StatusCode  
"Description`t" + $site.StatusDescription
```

The preceding command outputs the status code and description, as shown in the following image:



```
PS C:\> "The Site Status Code is`t" + $site.StatusCode  
"Description`t" + $site.StatusDescription  
The Site Status Code is 200  
Description OK
```

PS C:\>

Output

Now, run the following command:

```
Invoke-WebRequest -Uri http://www.Bing.com
```

The `Invoke-WebRequest` cmdlet sends http(s), FTP, and file requests and returns the collections of forms, links, images, and related HTML, as shown in the following image:

```
StatusCode      : 200
StatusDescription : OK
Content         : <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html lang="nl"
                  xml:lang="nl" xmlns="http://www.w3.org/1999/xhtml"><script type...
RawContent      : HTTP/1.1 200 OK
                  Vary: Accept-Encoding
                  Content-Length: 59399
                  Cache-Control: private, max-age=0
                  Content-Type: text/html; charset=utf-8
                  P3P: CP="NON UNI COM NAV STA LOC CURa DEVa PSAa PSDa OUR IND"...
Forms           : {sb_form}
Headers         : {[Vary, Accept-Encoding], [Content-Length, 59399], [Cache-Control, private,
                  max-age=0], [Content-Type, text/html; charset=utf-8]...}
Images          : {@{innerHTML=; innerText=; outerHTML=<IMG id=id_p class="b_icon id_avatar"
                  style="DISPLAY: none" src="data:image/gif;base64,R0lGODlhAQABIAAAAAAP//yH5BAEAA
                  AEALAAAAAABAAEAAAIBTA7">; outerText=; tagName=IMG; id=id_p; class=b_icon
                  id_avatar; style=DISPLAY: none; src=data:image/gif;base64,R0lGODlhAQABIAAAAAAP//yH5BAEAAEALAAAAAABAAEAAAIBTA7}}
InputFields     : {@{innerHTML=; innerText=; outerHTML=<INPUT spellcheck=false id=sb_form_q
                  title="Voer de zoekterm in" class=b_searchbox maxLength=1000 name=q
                  autocomplete="off" autocorrect="off" autocapitalize="off">; outerText=;
                  tagName=INPUT; spellcheck=false; id=sb_form_q; title=Voer de zoekterm in;
                  class=b_searchbox; maxLength=1000; name=q; autocomplete=off; autocorrect=off;
                  autocapitalize=off}, @{innerHTML=; innerText=; outerHTML=<INPUT tabIndex=0
                  id=sb_form_go title=Zoeken class=b_searchboxSubmit type=submit value="Submit
                  Querv" name=oo>; outerText=; tagName=INPUT; tabIndex=0; id=sb_form_oo}
```

Let's try to log in to Facebook using the `Invoke-WebRequest` cmdlet:

```
$request = Invoke-WebRequest
'https://www.facebook.com/login.php' -SessionVariable lgn
$Forms = $request.Forms[0]
$Forms.Fields
```

This returns the internal field names. We can do the same using the **View Source** option in the browser. Run the following command:

```
$Forms.Fields['email'] = 'UserID'
$Forms.Fields['Password'] = 'Password'
```

Using the preceding code, we can pass the username and password credentials.

Using the session variable, we will post the form action. Run the

following command:

```
$login =Invoke-WebRequest -Uri ("https://www.facebook.com" +  
$form.Action) -WebSession $lgn -Method POST -Body $Forms.Fields
```

The output of the session variable `$lgn` is shown in the following image:

```
Headers : {}  
Cookies : System.Net.CookieContainer  
UseDefaultCredentials : False  
Credentials :  
Certificates :  
UserAgent : Mozilla/5.0 (Windows NT; Windows NT 6.1; en-US) WindowsPowerShell/5.0.10100.0  
Proxy :  
MaximumRedirection : -1
```

Now, let's take a look at the `POST` action's output:

```
$login =Invoke-WebRequest -Uri ("https://www.facebook.com" +  
$form.Action) -WebSession $lgn -Method POST -Body $Forms.Fields  
$fb = $login.Forms[0]  
$fb
```

The output is illustrated in the following image:

```
PS C:\> $login =Invoke-WebRequest -Uri ("https://www.facebook.com" + $form.Action) -WebSession $lgn -Method POST -Body $Forms.Fields  
$fb = $login.Forms[0]  
PS C:\> $fb  


| ID         | Method | Action                                             | Fields                                                        |
|------------|--------|----------------------------------------------------|---------------------------------------------------------------|
| login_form | post   | https://www.facebook.com/login.php?login_attempt=1 | {[lsd, AVolEwt], [email, ], [pass, ], [u_0_v, A anmelden]...} |


```



Executing the following command returns the e-mail ID:

```
$Forms.Fields
```

However, the password will not be shown here. So, you can run the following command, which will show you the information:

```
$login.RawContent
```

In order to play with **Representation State Transfer (REST)**,

PowerShell has the `Invoke-RestMethod` cmdlet:

```
help Invoke-RestMethod -Detailed
```

The cool feature of Windows PowerShell is that it will format the return type based on the output response. If the output response is in RSS or ATOM, the XML nodes will be in those formats; however, for JSON and XML, they will be deserialized as objects.

In contrast to `Invoke-WebRequest`, we use `Invoke-RestMethod` because we obtain the contents, and it excludes the headers.

The following code returns the contents; this is not a friendly output:

```
Invoke-RestMethod -Uri  
http://blogs.technet.com/b/wikinijas/rss.aspx
```

So, we can play with this to get the information required, as follows:

```
Invoke-RestMethod -Uri  
http://blogs.technet.com/b/wikinijas/rss.aspx |  
Select Creator , link
```

We can explore the methods and properties and get the output, as follows:

```
Invoke-RestMethod -Uri  
http://blogs.technet.com/b/wikinijas/rss.aspx | Select Creator  
, PubDate
```

Downloading files from the Internet

Using PowerShell, we can download files from the Internet. Let's consider that you need to download a bunch of files from various portals and process them for reporting. Windows PowerShell is an easy way to achieve this. Run the following command:

```
$param = @{  
    URI = 'http://powershell.com/Mastering-PowerShell.pdf'  
    Outfile = 'C:\Temp\Mastering-PowerShell.pdf'  
}  
Invoke-RestMethod @param
```

Now, we will find the `Mastering-PowerShell.pdf` file in our `C:\Temp` folder.

Alternatively, we can use `System.Net.WebClient` as well. Run the following command:

```
$wc = New-Object System.Net.WebClient  
$wc.DownloadFile('http://powershell.com/Mastering-  
PowerShell.pdf' , 'C:\Temp\Mastering-PowerShell.pdf')
```

As we will use this for multiple downloads, naming each file is not a good option. So, we will tweak it a little more and save the file with the default name. Run the following command:

```
$Images = 'Url1/image1.jpg , url1/image2.jpg'  
$Dir = 'C:\Temp\'  
$wc = New-Object System.Net.WebClient  
foreach ($sourceFile in $Images){  
    $sourceFileName =  
    $sourceFile.Substring($sourceFile.LastIndexOf('/')+1)  
    $targetFileName = $targetDirectory + $sourceFileName  
    $wc.DownloadFile($sourceFile, 'C:\Temp')  
}
```

Reading a file from the Internet

Reading files from the Internet using PowerShell is a good option, but we don't want to open the browser all the time. We can read RSS and obtain information related to it. We did this using the `Invoke-WebRequest` cmdlet in the previous topic. Now, we will use the `WebClient` class. Run the following command:

```
$url = "http://blogs.technet.com/b/wikininjas/rss.aspx"
```

The URL we have picked up is TechNet Wiki Ninjas RSS.

In the following code, we will typecast it with XML:

```
[xml]$xml = (New-object  
System.Net.WebClient).DownloadString($url)
```

Let's consider the output of XML; take a look at the following command

and the subsequent image:

```
$url = "http://blogs.technet.com/b/wikininjas/rss.aspx"
$xml = [xml](New-object System.Net.WebClient).DownloadString($url)
$xml
```

A screenshot of a PowerShell window. The command is \$url = "http://blogs.technet.com/b/wikininjas/rss.aspx"; \$xml = [xml](New-object System.Net.WebClient).DownloadString(\$url); \$xml. The output shows the XML structure of the RSS feed, including the XML declaration, root element 'rss', and a 'channel' element with a 'stylesheet' attribute pointing to a URL. A red speech bubble in the top right corner says 'RSS - Easy to play now!'. The PowerShell window has a dark blue background.

To read the description of the given RSS feed, we will execute the following code:

```
$xml.rss.channel.description
```

This returns the following:

```
Wiki Ninjas is a group of authors who celebrate and evangelize the community catalyst and social-authoring excellence that is called TechNet Wiki.
```

To read the creators, we will execute the following code:

```
$url = "http://blogs.technet.com/b/wikininjas/rss.aspx"
$xml = [xml](New-object System.Net.WebClient).DownloadString($url)
$xml.rss.channel | %{$_.item} | ? {[DateTime]$_.pubDate -gt (Get-Date).AddMonths(-3)} | FT creator
```

You can add `link`, `pubdate`, and perform many other tasks. To see a post that was published one week earlier, run the following command:

```
$url = "http://blogs.technet.com/b/wikininjas/rss.aspx"
$xml = [xml](New-object System.Net.WebClient).DownloadString($url)
$xml.rss.channel | %{$_.item} | ? {[DateTime]$_.pubDate -gt (Get-Date).AddDays(-7)} | FT creator , pubdate -AutoSize
```

Take a look at the following image:

```
PS C:\> $url = "http://blogs.technet.com/b/wikinijas/rss.aspx"
[xml]$xml = (New-object System.Net.WebClient).DownloadString($url)
$xml.rss.channel | %{$_.item} | ? {[DateTime]$_.pubDate -gt (Get-Date).AddDays(-7)} | FT creator , pubdate -AutoSize
creator          pubDate
-----          -----
Durval Ramos    Fri, 31 Jul 2015 02:03:08 GMT
Ed Price - MSFT Tue, 28 Jul 2015 07:00:00 GMT
Ed Price - MSFT Sun, 26 Jul 2015 17:10:00 GMT
XAML guy        Sat, 25 Jul 2015 23:07:23 GMT
Ed Price - MSFT Sat, 25 Jul 2015 00:35:00 GMT
```

Exploring web services

A web service is generally an XML-based program. Using this, we can exchange data over a network. A web service is nothing but an application component, which works on an open protocol, can be used by other applications, and is reusable.

Web services allow interaction between applications and use standardized XML. Simply put, it's a combination of XML and HTTP. The following components work for all web services:

- **Simple Object Access Protocol (SOAP)**
- **Universal Description, Discovery, and Integration (UDDI)**
- **Web Services Description Language (WSDL)**

In this topic, we will demonstrate the use of web services and complex web services using Windows PowerShell.

Using web services

`New-WebServiceProxy` is a cmdlet that allows us to create a proxy object of any given valid web service.

For our example, we will use the following web service:

<http://www.webservicex.net/CurrencyConvertor.asmx?WSDL>

Run the following command:

```
$var = New-WebServiceProxy -Uri  
http://www.webservicex.net/CurrencyConvertor.asmx?WSDL  
$var.ConversionRate
```

This returns the overload definitions of the conversion rate method, as shown in the following image:

```
PS C:\> $var = New-WebServiceProxy -Uri http://www.webservicex.net/CurrencyConvertor.asmx?WSDL
$var.ConversionRate
overloadDefinitions
double ConversionRate(Microsoft.PowerShell.Commands.NewWebServiceProxy.AutogeneratedTypes.WebServiceProxy3et_CurrencyConvertor_asmx_WSDL
.Currency FromCurrency, 1 Microsoft.PowerShell.Commands.NewWebServiceProxy.AutogeneratedTypes.WebServiceProxy3et_CurrencyConvertor_asmx_WSDL.Currency ToCurrency) 2
```

Points marked in the image we just saw are explained as follows:

- 1: The first argument should be `FromCurrency`
- 2: The second argument should be `ToCurrency`

Here is a simple example of this:

```
$var = New-WebServiceProxy -Uri
http://www.webservicex.net/CurrencyConvertor.asmx?WSDL
$var.ConversionRate('Eur', 'US')
```

This returns the output as 1.0979—the current conversion value.

How do we know the currency codes? Just open the following URL in a browser:

<http://www.webservicex.net/CurrencyConvertor.asmx?WSDL>

This shows the output as illustrated in the following image:

```
</s:element>
- <s:simpleType name="Currency">
  - <s:restriction base="s:string">
    <s:enumeration value="AFA"/>
    <s:enumeration value="ALL"/>
    <s:enumeration value="DZD"/>
    <s:enumeration value="ARS"/>
    <s:enumeration value="AWG"/>
    <s:enumeration value="AUD"/>
    <s:enumeration value="BSD"/>
    <s:enumeration value="BHD"/>
    <s:enumeration value="BDT"/>
    <s:enumeration value="BBB"/>
    <s:enumeration value="BZD"/>
    <s:enumeration value="BMD"/>
    <s:enumeration value="BTN"/>
    <s:enumeration value="BOB"/>
    <s:enumeration value="BWP"/>
    <s:enumeration value="BRL"/>
    <s:enumeration value="GBP"/>
    <s:enumeration value="BND"/>
    <s:enumeration value="BIF"/>
    <s:enumeration value="XOF"/>
    <s:enumeration value="XAF"/>
    <s:enumeration value="KHR"/>
```

Currency Values /
Codes

Similarly, we can use the **GlobalWeather** web service to retrieve information:

<http://www.webservicex.net/globalweather.asmx?wsdl>

Run the following command:

```
$weather = New-WebServiceProxy -uri
http://www.webservicex.com/globalweather.asmx?WSDL
([XML]$weather.GetWeather('Amsterdam', 'Netherlands')).CurrentWe
ather
```

The output is illustrated in the following image:

```

Location : Amsterdam Airport Schiphol, Netherlands (EHAM) 52-18N 004-46E -2M
Time    : Jul 31, 2015 - 06:55 AM EDT / 2015.07.31 1055 UTC
Wind    : Variable at 3 MPH (3 KT):0
Visibility : greater than 7 mile(s):0
SkyConditions : mostly clear
Temperature : 62 F (17 C)
DewPoint   : 44 F (7 C)
RelativeHumidity : 51%
Pressure    : 30.12 in. Hg (1020 hPa)
Status      : Success

```

Web services are usually limited to XML and SOAP, but they may be in the JSON format as well. So, it's worth considering an example that returns results in the JSON format.

Let's use an Apple iTunes URL, which returns an output in the JSON format:

<http://itunes.apple.com/search?term=metallica>

If we open this URL in the browser, we will see the output as shown in the following image:



```
{
  "resultCount":50,
  "results": [
    {"wrapperType":"track", "kind":"song", "artistId":3996865, "collectionId":579372950, "trackId":579373079, "artistName":"Metallica", "collectionCensoredName":"Metallica", "trackCensoredName":"Enter Sandman", "artistViewUrl":"https://itunes.apple.com/us/artist/metallica/id579372950?i=579373079&uo=4", "trackViewUrl":"https://itunes.apple.com/us/album/enter-sandman/id579372950?i=579373079&uo=4", "previewUrl":"http://a38.phobos.apple.com/us/r30/Music7/v4/bd/e4/bdfde4e-5407-9bb0-e632-edbf079bed21/mzaf_907706799096684396.plus.aac.p4/thumb/Music/v4/9a/41/03/9a410309-7dbe-fce4-a6be-43101f0a1352/0093624986553.jpg/30x30bb-85.jpg", "artworkUrl160":"http://is2.mzstatic.com/image/thumb/Music/v4/9a/41/03/9a410309-7dbe-fce4-a6be-43101f0a1352/0093624986553.jpg/60x60bb-85.jpg", "artworkUrl100":"http://is3.mzstatic.com/image/thumb/Music/v4/9a/41/03/9a410309-7dbe-fce4-a6be-43101f0a1352/0093624986553.jpg/100x100bb-85.jpg", "collectionPrice":7.99, "trackPrice":1.29, "releaseDate":"1991-08-12T07:00:00Z", "collectionExplicitness":"notExplicit", "trackExplicitness":"notExplicit", "trackCount":12, "trackNumber":1, "trackTimeMillis":331560, "country":"USA", "currency":"USD", "primaryGenreName":"Rock", "radioStationUrl":"isStreamable":true},
    {"wrapperType":"track", "kind":"song", "artistId":3996865, "collectionId":579372950, "trackId":579373086, "artistName":"Metallica", "collectionCensoredName":"Metallica", "trackCensoredName":"Nothing Else Matters", "artistViewUrl":"https://itunes.apple.com/us/artist/metallica/id579372950?i=579373086&uo=4", "trackViewUrl":"https://itunes.apple.com/us/album/nothing-else-matters/id579372950?i=579373086&uo=4", "previewUrl":"http://a38.phobos.apple.com/us/r30/Music7/v4/3c/e7/ac/3ce7ac90-4834-151a-667e-2259c7e7bd38/previewUrl130":"http://is5.mzstatic.com/image/thumb/Music/v4/9a/41/03/9a410309-7dbe-fce4-a6be-43101f0a1352/0093624986553.jpg/30x30bb-85.jpg", "artworkUrl130":"http://is5.mzstatic.com/image/thumb/Music/v4/9a/41/03/9a410309-7dbe-fce4-a6be-43101f0a1352/0093624986553.jpg/60x60bb-85.jpg", "artworkUrl100":"http://is3.mzstatic.com/image/thumb/Music/v4/9a/41/03/9a410309-7dbe-fce4-a6be-43101f0a1352/0093624986553.jpg/100x100bb-85.jpg", "collectionPrice":7.99, "trackPrice":1.29, "releaseDate":"1991-08-12T07:00:00Z", "collectionExplicitness":"notExplicit", "trackExplicitness":"notExplicit", "discCount":1, "discNumber":1, "trackCount":12, "trackNumber":8, "trackTimeMillis":388733, "country":"US"
  ]
}
```

Here, we will not use the `Invoke-RestMethod` cmdlet because it formats the data by default. Instead, we will use the `Invoke-WebRequest` cmdlet for the `ConvertFrom-Json` cmdlet:

```
$Json = Invoke-WebRequest -Uri "http://itunes.apple.com/search?term=metallica"
```

\$Json

Take a look at the following image:

```
StatusCode      : 200
StatusDescription : OK
Content         :
RawContent      :
Forms           :
Headers          :
Images          :
Inputfields     :
Links            :
ParsedHtml       : mshtml.HTMLDocumentClass
RawContentLength : 76958

{
    "resultCount":50,
    "results": [
        {"wrapperType":"track", "kind":"song", "artistId":3996865, "collectionId":579372950, "trackId":579373079, "artistName":"Metallica",
        "collectionName":"Metallica", ...}
    ]
}

x-apple-jingle-correlation-key: TIGLVAHYXJEPAMBmjGYB2LTHAE
x-apple-translated-wo-url: /webobjects/MZStoreServices.woa/ws/wsSearch?term=metallica&urlDesc=
x-apple-orig-url: http://it...
: {}

: {[x-apple-jingle-correlation-key, TIGLVAHYXJEPAMBmjGYB2LTHAE], [x-apple-translated-wo-url,
/webobjects/MZStoreServices.woa/ws/wsSearch?term=metallica&urlDesc=], [x-apple-orig-url,
http://itunes.apple.com/search?term=metallica], [x-apple-application-site, ST1]...}

: {}

: {}

: {}

: mshtml.HTMLDocumentClass

: 76958
```

Now, we will play with the PowerShell code to convert this content into a readable format:

```
$Json = Invoke-WebRequest -Uri "http://itunes.apple.com/search?
term=metallica"
($Json | ConvertFrom-Json).Results | Select WrapperType,
trackName | Select -First 25
```

Now, consider the following image:

```
wrapperType trackName
-----
track      Enter Sandman
track      Nothing Else Matters
track      The Unforgiven
track      One
track      Wherever I May Roam
track      Sad But True
track      For Whom the Bell Tolls
track      The Day That Never Comes
track      Master of Puppets
track      Fade to Black
track      Fuel
track      Turn the Page
track      Cyanide
track      Don't Tread On Me
track      All Nightmare Long
track      Whiskey In the Jar
track      The Unforgiven III
track      Holier Than Thou
track      My Friend of Misery
track      Of Wolf and Man
track      Through the Never
track      The Unforgiven II
track      The God That Failed
track      I Disappear
track      King Nothing
```

Converted JSON to
readable format and
selected first 25
results

Alternatively, we can use the `Invoke-RestMethod` cmdlet and do the same:

```
$Json = Invoke-RestMethod -Uri "http://itunes.apple.com/search?
term=metallica"
$Json.results | Select WrapperType , trackName | Select -First
25
```

Building web services

In this exercise, we will take a look at the steps required to build web services. For this, we will use the following environments:

- Windows Server 2012
- Visual Studio 2013
- The installed IIS server

We configured IIS in WMF5Node03 to demonstrate PowerShell web access; now, let's connect to the same box and build a small web service. Perform the following steps:

1. Open **Visual Studio**.
2. Select **.NET framework 3.5**.
3. Then, select **ASP.NET Web Service**.
4. Select **HTTP** and name your site.
5. In our scenario, we will name it `http://localhost/website`.

Visual Studio builds the basic web service. You can run the following command:

```
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
' To allow this Web Service to be called from script, using
ASP.NET AJAX, uncomment the following line.
' <System.Web.Services.ScriptService()> _
<WebService(Namespace:="http://tempuri.org/")> _
<WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1)> _
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerate
d()> _
Public Class Service
    Inherits System.Web.Services.WebService
    <WebMethod()> _
    Public Function HelloWorld() As String
        Return "This is a Basic and Default Web Service!"
    End Function
End Class
```

Take a look at the following image:

Service

The following operations are supported. For a formal definition, please review the [Service Description](#).

- [HelloWorld](#)

This web service is using <http://tempuri.org/> as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

Each XML Web service needs a unique namespace in order for client applications to distinguish it from other services on the Web. <http://tempuri.org/> is available for XML Web services that are under development, but published XML Web services should use a more permanent namespace.

Your XML Web service should be identified by a namespace that you control. For example, you can use your company's Internet domain name as part of the namespace. Although many XML Web service namespaces look like URLs, they need not point to actual resources on the Web. (XML Web service namespaces are URLs.)

For XML Web services creating using ASP.NET, the default namespace can be changed using the WebService attribute's Namespace property. The WebService attribute is an attribute applied to the class that contains the XML Web service methods. Below is a code example that sets the namespace to "http://microsoft.com/webservices":

C#

```
[WebService(Namespace="http://microsoft.com/webservices/")]
public class MyWebService {
    // implementation
}
```

Visual Basic

```
<WebService(Namespace:="http://microsoft.com/webservices/")> Public Class MyWebService
    ' implementation
End Class
```

C++

```
[WebService(Namespace="http://microsoft.com/webservices/")]
public ref class MyWebService {
    // implementation
};
```

For more details on XML namespaces, see the W3C recommendation on [Namespaces in XML](#).

For more details on WSDL, see the [WSDL Specification](#).

For more details on URIs, see [RFC 2396](#).

So, now we have the URL; we simply changed the Hello World text to something else, as follows:

<http://localhost/website/Service.asmx>

Okay, spin up PowerShell now!

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The session starts with the command `PS C:\> $local = New-WebServiceProxy -Uri http://localhost/website/Service.asmx?wsdl`. A red circle labeled "1" is placed over the "Administrator" title bar. The output shows the properties of the \$local variable, which includes the URL and various proxy settings. A red circle labeled "2" is placed over the first few properties. The next command, `PS C:\> $local`, is shown with a red circle labeled "3" over the cursor. The final command, `PS C:\> $local.HelloWorld()`, is shown with a red circle labeled "4" over the cursor. The output of the last command is "Hello World".

```
PS C:\> $local = New-WebServiceProxy -Uri http://localhost/website/Service.asmx?wsdl
PS C:\> $local
2
SoapVersion : Default
AllowAutoRedirect : False
CookieContainer :
ClientCertificates :
EnableDecompression : False
UserAgent : Mozilla/4.0 (compatible; MSIE 6.0; MS Web Services Client Protocol
4.0.30319.34014)
Proxy :
UnsafeAuthenticatedConnectionSharing : False
Credentials :
UseDefaultCredentials : False
ConnectionGroupName :
PreAuthenticate : False
Url : http://localhost/website/Service.asmx
RequestEncoding :
Timeout : 100000
Site :
Container :

3
PS C:\>
PS C:\> $local.HelloWorld()
Hello World
4
PS C:\>
```

Points marked in the figure are explained as follows:

- 1: At the end of the URL, append ?wsdl

- 2: Assign it to the variable named \$local
- 3: Invoke the function HelloWorld()
- 4: It returns the default output

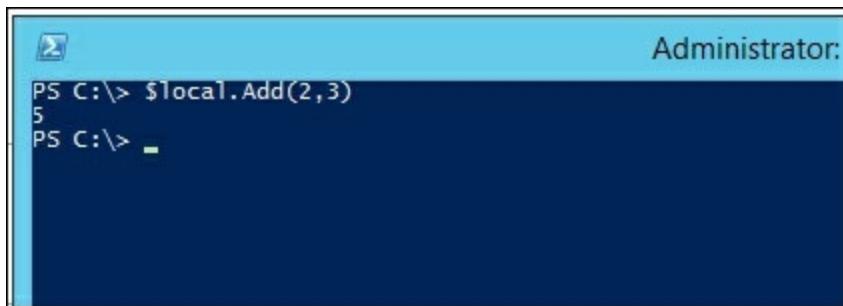
Invoke after doing text changes in the code which return your newly updated text

Now, in Service.vb, add the following code:

```
<WebMethod()> _
Public Function Add(a, b) As Int32
    Return a + b
End Function
```

We can add two values in Windows PowerShell:

```
$local.Add(2,3)
```



Create your own web service and consume Windows PowerShell to use it.

Exploring the REST API

REST is the abbreviated form of Representational State Transfers. Using this, we can connect to the server from a client over HTTP and manipulate resources. In short, REST is a software architectural model commonly used on the World Wide Web.

Take a look at the help document of the `Invoke-RestMethod` method which sends an HTTP or HTTPS request to a RESTful web service.

Using the Azure REST API in PowerShell

Managing **Azure** is easy using the REST API and PowerShell. No need for employees clicking Azure machines to retrieve information from the Azure portal. Many IT professionals use the Azure module to automate tasks in Azure, such as building, restarting, and removing the server, and many more. Before we start using the REST API, let's explore a few Azure cmdlets.

As with any other module, we need to import the Azure module as well. A few errors in the Azure module might be misleading, but we can safely ignore these for now.

The first step is to get the Azure publish settings file. Run the following command:

```
Import-Module Azure -verbose  
Get-AzurePublishSettingsFile
```

The preceding command opens up your default browser and downloads the file. The next step is to import the Azure file. Run the following command:

```
Import-AzurePublishSettingsFile -PublishSettingsFile  
C:\Temp\file
```

That's it! Now, we can play with the Azure cmdlets.

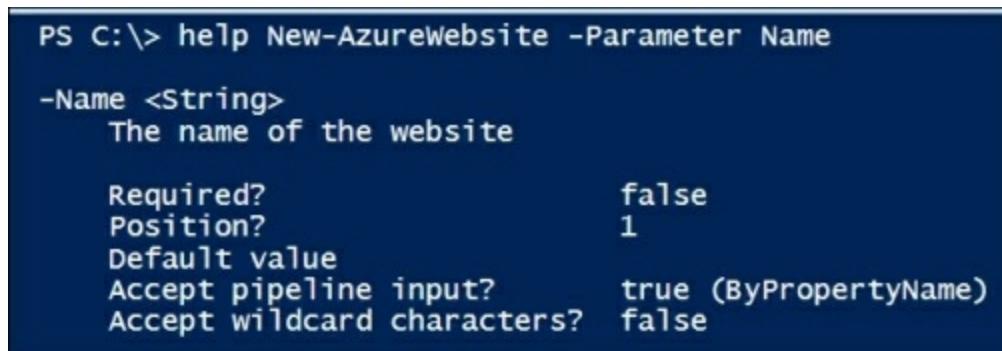
Approximately 737 cmdlets are available in the Azure module. Explore the help and automate your tasks. As this topic is about the Azure REST API, let's create a website for us to walk through. Run the following command:

```
New-AzureWebsite -Name "MyAzureSiteDemo" -Verbose
```

Well! We've now created an Azure website. Very much in a human readable style which is PowerShell Verb-Noun with parameters. You may wonder, what does the preceding command do? It simply creates a new Azure website. Indeed, you can create multiple sites; for this, we need to consider the `Name` property. Run the following command:

```
help New-AzureWebsite -Parameter Name
```

Now, take a look at the following image:



```
PS C:\> help New-AzureWebsite -Parameter Name

-Name <String>
    The name of the website

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

This accepts pipeline inputs. The following code shows the method to process multiple sites.

```
"Site1" , "Site2" | %{
    New-AzureWebsite -Name $_ -Verbose
}
```

Now, let us see the **Kudu** project from Git—It's huge, so you need to go through the details using the following link:

<https://github.com/projectkudu/kudu>

With reference to this link, we will try to fetch the version with the site's

last modified time stamp, as follows:

```
$site = Get-AzureWebsite -Name MyAzureSiteDemo
$username = $site.PublishingUsername
$password = $site.PublishingPassword
$base64AuthInfo =
[Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("
{0}:{1}" -f $username,$password)))
$api = "https://$(($site.Name).scm.azurewebsites.net/api"
$skudu = Invoke-RestMethod -Uri "$api/environment" -Headers
@{Authorization=("Basic {0}" -f $base64AuthInfo)} -Method GET
$skudu
```

Output of the command is shown in the following figure:

The screenshot shows a PowerShell command being run in a terminal window. The command is as follows:

```
PS C:\> $site = Get-AzureWebsite -Name MyAzureSiteDemo ①
$username = $site.PublishingUsername ②
$password = $site.PublishingPassword ③
$base64AuthInfo = [Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("
{0}:{1}" -f $username,$password))) ④
$api = "https://$(($site.Name).scm.azurewebsites.net/api" ⑤
$skudu = Invoke-RestMethod -Uri "$api/environment" -Headers @
{Authorization=("Basic {0}" -f $base64AuthInfo)} -Method GET ⑥
$skudu
```

Below the command, the output is displayed:

version	siteLastModified
47.40727.1709.0	2015-07-31T16:42:45.770000Z ⑦

Red numbers 1 through 8 are overlaid on the command and output to indicate specific steps or components of the code.

The code performs eight actions to fetch the information we need:

- 1: First, it obtains the azure website information as Object.
- 2: Then, it gets the username and password of the site (This is a property of the `Get-AzureWebSite` cmdlet).
- 3: By typecasting the code, it will convert the authentication to base64 encoding.
- 4: Here, we used the API URL—this is REST (Note that the site we created doesn't have the GIT resource mapped—it's just a dummy site).
- 5: Now, the power of the `Invoke-WebRequest` cmdlet comes into the picture—we have come across this cmdlet and its features in the previous topic.
- 6: In the preceding code, we used an environment in the URL. If you are interested in exploring and doing more, refer to the following link and proceed further:

<https://github.com/projectkudu/kudu/wiki/REST-API>

- 7: In this step, we used the Authorization headers. This parameter is of the `<IDictionary>` type and can't be used for the user agent or cookie. Just specify the header like shown in the code. Here we are passing the authorization value as `Basic` and `{0}` represents the place holder. The value of the `$base64AuthInfo` variable will be passed like `Basic <$base64AuthInfo>`.
- 8: Finally, we retrieved the output, as you saw in the figure.

However, this is not enough for the infrastructure, so we need to do more! We need to jump-start **Azure Resource Manager (ARM)**. We do have a module for this where we can manage the Azure environment—behind the scenes, it invokes the REST API.

If you are a DevOps looking for an ARM and **Application Lifecycle Management (ALM)** integration, use the following link:

<https://www.microsoftvirtualacademy.com/en-us/training-courses/azure-resource-manager-devops-jump-start-8413>

Note

As the following modules are separated:

- `AzureResourceManager`
- `AzureServiceManagement`

To explore resource manager, we should first run the following command:

```
Switch-AzureMode -Name AzureResourceManager
#or
Switch-AzureMode -Name AzureServiceManagement
```

Read the help document, explore Azure, and use the REST API.

Exploring JSON

JavaScript Object Notation (JSON) is a lightweight, text-based, open standard that is designed for data interchange. It is language-independent, with parsers available for Windows PowerShell.

The JSON format is often used to serialize and transmit structured data over a network connection. It is used primarily to transmit data between a server and a web application, as an alternative to XML.

You can use the following link to validate the JSON format:

<http://jsonformatter.curiousconcept.com/>

Here is a simple example of the JSON Format:

```
{  
    "Title": "Mr.",  
    "Name": "Scripting"  
}
```

Windows PowerShell has two cmdlets: `ConvertFrom-Json` and `ConvertTo-Json`. Take a look at the following image:

PS C:\> Get-Command -Name *JSON*			
CommandType	Name	Version	Source
Cmdlet	ConvertFrom-Json	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	ConvertTo-Json	3.1.0.0	Microsoft.PowerShell.Utility

As the name reads, it converts to and from. Let's take a look at how the following code works:

```
$json = @"  
{  
    "Title": "Mr.",  
    "Name": "Scripting"  
}
```

```
"@"
$json | ConvertFrom-Json
```

Now, consider the following image:

A screenshot of a PowerShell window. The command entered is:

```
PS C:\> $json = @"
{ "Title": "Mr.", "Name": "Scripting"
} "@
$json | ConvertFrom-Json
```

The output is:

Title	Name
Mr.	Scripting

Annotations with arrows point to different parts of the code and output:

- An arrow points from the JSON block inside the here-string to a callout box labeled "JSON inside the here string!".
- An arrow points from the pipeline operator "| ConvertFrom-Json" to a callout box labeled "Convert".
- An arrow points from the output table to a callout box labeled "Output".

The steps performed in the figure we just saw is explained as follows:

- We have passed the JSON formatted code inside the here-string
- Using pipeline we are converting the JSON to a `PSCustomObject`
- Where `Title` and `Name` will be a `NoteProperty` with the values

Take a look at the following image:

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=Scripting
Title	NoteProperty	string Title=Mr.

Using JSON cmdlets, we can manipulate the contents as required:

```
$json = @"
{
    "User1": { "Name": "Chen", "Role": "SharePoint IT Pro" },
    "User2": { "Name": "Keanu" }
}
```

```
"@  
$data = $json | ConvertFrom-Json  
$data.User1  
$data.User2
```

Now, consider the following image:

Name	Role
Chen	SharePoint IT Pro
Keanu	

It's easy to read a JSON file in PowerShell. Let's consider that we have a JSON file, as follows:

```
{  
    "1": "One",  
    "2": "Two",  
    "3": "Three"  
}  
$data = Get-Content C:\Temp\JSON.json -Raw | ConvertFrom-Json  
$data.psobject.Properties.Name
```

The preceding code will provide the output as 1, 2, and 3.

Summary

In this chapter, we explored a few, good features of Windows PowerShell, especially the ones related to web technologies. Using this, we can automate tasks such as installing or uninstalling PowerShell Web Access and configuring it, tasks related to web requests, and consuming web services to perform some tasks. As JSON is considered to be an alternative to XML, and is language-independent, we can use it in Windows PowerShell and perform our web-based queries or other tasks easily and effectively.

In the next chapter, we will discuss the usage of **Application Programming Interfaces (API)** in Windows PowerShell.

Chapter 5. Exploring Application Programming Interface

Application Programming Interface (API) is a set of protocols, routines, and tools used to build software applications. The main purpose of API is data exchange and integration, whereas **Software Development Kit (SDK)** is customization. The type of API varies based on the data exchange and transport mechanism. The transport mechanism may be web-based, source code, or binary function.

Following are the most common types of API:

- Web Services API
 - This is an API that is highly useful in up and coming technologies. It is hypertext-based and the standards used are REST, SOAP, XML-RPC, and JSON-RPC.
 - It is commonly used in web applications and **service-oriented architecture (SOA)**.
- Source code API
 - This offers libraries of objects, classes, and so on
 - It is commonly used in development projects
 - The standards used vary based on the platform, such as .NET or J2EE

In this chapter, we will cover the following topics:

- Exploring API using PowerShell
- The **Exchange Web Services (EWS)** API for managing online exchange, along with a demo involving purging items in mailbox folders and deleting items
- The Lync 2013 / Skype for Business client-side API—installing the Lync API, exploring the client settings, automating test calls, and holding an IM conversation with contacts
- Client server object model for managing **SharePoint Online**—installing client-side SDK
- Exploring **client-side object model (CSOM)**

- Manipulating SharePoint lists using CSOM in PowerShell

Developers use APIs to accomplish many tasks; using PowerShell, we can only achieve our tasks to some extent. For example, let's consider a scenario where we need to interact with a Windows API; to achieve this, we can use the Windows API. Most IT professionals don't use API considering the fact that it's meant for development. However, we can leverage APIs in Windows PowerShell to perform administration as well as development tasks.

In the following example, we will try to use the Windows API in PowerShell as a jump start. Let's take a look at the PowerShell way of playing with the `User32.dll` file.

Note

You can also refer to <https://msdn.microsoft.com/en-us/library/windows/desktop/ms633548%28v=vs.85%29.aspx>

The C++ code used is as follows:

```
BOOL WINAPI ShowWindow(
    _In_ HWND hWnd,
    _In_ int nCmdShow
);
```

In Windows PowerShell, we will use the `Add-type` command for this exercise. This is not a big deal! It's a pretty old concept. However, to begin, we need to know how PowerShell loads the standard assembly. Later in this topic, we will examine the managed assemblies as well. Run the following command:

```
Help Add-Type -Detailed
```

Yes, as like all other cmdlets, we need to first read the help document before we proceed further. Using the preceding result, we can explore the parameters and related information. Now, we will build a PowerShell script that performs a few actions on Windows, which is as follows:

```
Function Set-Window {
```

```

Param(
    [Parameter(Mandatory = $true)]
    $ID
)
$code = @"
[DllImport("user32.dll")]
public static extern bool ShowWindowAsync(IntPtr hWnd, int nCmdShow);
"@
$demo = Add-Type -MemberDefinition $code -Name "Demo" -
Namespace Win32Functions -PassThru
$demo::ShowWindowAsync((Get-Process -id $ID).MainWindowHandle
, 2)
}
Set-Window -ID $pid

```

How does this work? We created a PowerShell function named `Set-Window`, which accepts a parameter called `ID`. Run the following command:

```

$code = @"
[DllImport("user32.dll")]
public static extern bool ShowWindowAsync(IntPtr hWnd, int nCmdShow);
"@

```

Using the previous snippet of code inside the here-string, we will import the `user32.dll` file and consume the `ShowWinodAsync` function. This accepts the following two overloads:

- `hWnd`: An example of this is process information
- `nCmdShow`: This is a set of valid parameters listed in the following table:

Value	Meaning
<code>SW_FORCEMINIMIZE</code> 11	This minimizes a window even if the thread that owns the window is not responding. This flag should only be used while minimizing windows from a different thread.
<code>SW_HIDE</code> 0	This hides the window and activates another window.

SW_MAXIMIZE	This maximizes the specified window.
3	
SW_MINIMIZE	This minimizes the specified window and activates the next top-level window in the Z order.
6	
SW_RESTORE	This activates and displays the window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag while restoring a minimized window.
9	
SW_SHOW	This activates the window and displays it in its current size and position.
5	
SW_SHOWDEFAULT	This sets the show state based on the <i>SW_</i> value specified in the STARTUPINFO (https://msdn.microsoft.com/en-us/library/windows/desktop/ms686331(v=vs.85).aspx) structure passed to the CreateProcess (https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx) function by the program that started the application.
10	
SW_SHOWMAXIMIZED	This activates the window and displays it as a maximized window.
3	
SW_SHOWMINIMIZED	This activates the window and displays it as a minimized window.
2	
SW_SHOWMINNOACTIVE	This displays the window as a minimized window. This value is similar to SW_SHOWMINIMIZED, except that the window is not activated.
7	
SW_SHOWNA	This displays the window in its current size and position. This value is similar to SW_SHOW, except that the window is not activated.
8	
SW_SHOWNOACTIVATE	This displays a window in its most recent size and position. This value is similar to SW_SHOWNORMAL, except that the window is not activated.
4	
SW_SHOWNORMAL	This activates and displays a window. If the window is minimized

1

or maximized, the system restores it to its original size and position. An application should specify this flag while displaying the window for the first time.

To instantiate the `ShowWindowAsnc` function, we will use the `Add-Type` cmdlet and assign it to the `$demo` variable, as shown in the following command line:

```
$demo = Add-Type -MemberDefinition $code -Name "Demo" -  
Namespace Win32Functions -PassThru  
$demo::ShowWindowAsync((Get-Process -id $ID).MainWindowHandle ,  
2)  
Set-Window -ID $pid
```

As we used `$pid` while executing the function, the code will affect the current PowerShell console and minimize it. We can perform the advanced functions as required and build a PowerShell code with reference to the `User32.dll` file.

In this chapter, we will cover PowerShell and API use for Microsoft technologies such as SharePoint, Exchange, and Lync; so, for another short example of .NET, let's take a look at the speech API.

Note

Refer to the following URL for the `System.Speech` API:

<https://msdn.microsoft.com/en-us/library/gg145021%28v=vs.110%29.aspx>

Run the following command:

```
Add-Type -AssemblyName System.Speech  
$speak = New-Object System.Speech.Synthesis.SpeechSynthesizer  
$speak.Rate = 1  
$speak.Speak("Welcome to PowerShell 5.0")
```

The preceding code will speak out the text in the last line:

"Welcome to PowerShell 5.0"

This command will perform the following functions:

- It will load the `System.Speech` assembly
- It will create a `$speak` object
- It will control the speaking rate at 1
- It will speak out the text

Exploring API using PowerShell

We considered a basic example of using the Windows API. There are different ways to use APIs in PowerShell based on the requirements. Windows PowerShell has the feature of interacting with .NET DLL files as well. Let's consider that you have a code that simply performs addition operations that we can load in PowerShell and then explores the methods. But why are we discussing DLL now? Here is a comparison of API and DLL:

API	DLL
Abstract	Concrete
In this, the interface is implemented by the software program	This is a method of providing APIs
An API is an interface to the library of code	DLL is nothing but a library of code

In short, DLL is a file format and a way to use API.

Let's take a look at a demo where we will use a custom DLL file in Windows PowerShell. For this, we will choose a class library in Visual C# and build a code that will simply add two given integers.

The C# code is as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ClassLibrary2
```

```
{
    public class Class1
    {
        public static int sum(int a,int b)
        {
            return a + b;
        }
    }
}
```

Note that you can rename namespace and class, which helps others understand them. Here, we have selected the default options for demonstration purposes.

We can directly use this with the here-string, and with the help of the `Add-Type` cmdlet, we can call the `sum` function and add two integer values. Run the following command:

```
$code = @"
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ClassLibrary2
{
    public class Class1
    {
        public static int sum(int a,int b)
        {
            return a + b;
        }
    }
}"
Add-Type -TypeDefintion $code
[ClassLibrary2.Class1]::sum(34,45)
```

The output is illustrated in the following image:

```

PS C:\> $code = @"
1
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ClassLibrary2
{
    public class Class1
    {
        public static int sum(int a,int b)
        {
            return a + b;
        }
    }
}"@          2          3          4
Add-Type -TypeDefintion $code
[ClassLibrary2.Class1]::sum(34,45) 5
79 6

```

Points marked in the figure can be explained as follows:

- 1: Everything between @ and @" is the here-string
- 2: Here, we use the Add-Type cmdlet
- 3: Using the TypeDefinition parameter, we specify the source code
- 4: In the \$code variable, we have our source code in the here-string format
- 5: Here, we make use of the namespace, the class, and the sum method
- 6: Here, it returns the output value

Now, in Visual Studio, compile the code and find the DLL file; in our case, we have the DLL file in the Temp folder, so we can use the following snippet of code in PowerShell:

```

Import-Module
C:\Temp\ClassLibrary2\ClassLibrary2\bin\Debug\ClassLibrary2.dll
[ClassLibrary2.Class1]::sum(45,56)

```

Take a look at the following image:

```
PS C:\> Import-Module C:\Temp\ClassLibrary2\ClassLibrary2\bin\Debug\ClassLibrary2.dll
PS C:\> [ClassLibrary2.Class1]::sum(45,56)
101
PS C:\> |
```

Note that we have a `Class` keyword in WMF 5.0; we could use that. However, in PowerShell, we have a few constraints—refer to the WMF 5.0 release notes.

In the next topic, we will explore the EWS API for managing **Exchange Online**.

The EWS API for managing Exchange Online

The **Exchange Web Services (EWS)** Managed API provides a managed interface for developing .NET client applications that use EWS. Using the EWS Managed API, we can access most of the information stored in Office 365, Exchange Online, or the Exchange Server mailbox.

The EWS Managed API is now available as an open source project on GitHub. You can share your contributions and bug report bugs in GitHub.

Note

For the contributions, refer to the following link:

<https://github.com/OfficeDev/ews-managed-api/blob/master/CONTRIBUTING.md>

To report issues, refer to the following link:

<https://github.com/OfficeDev/ews-managed-api/issues>

For the MSDN documentation, refer to the following link:

<https://msdn.microsoft.com/en-us/library/office/dd633710%28v=exchg.80%29.aspx>

To download the EWS Managed API 2.2, refer to the following link:

<http://www.microsoft.com/en-us/download/details.aspx?id=42951>

The installation of the EWS Managed API is as simple as a click—click and it's done! For an easy reference and quick demo, we moved the DLL file to the C:\Temp\ews folder.

One of the easiest methods to explore the items in the mailbox folder is using the MSOnline module.

To connect to Exchange Online, we will use the following snippet of code:

```
Import-Module MSOnline
$O365Cred = Get-Credential
$O365Session = New-PSSession -ConfigurationName
Microsoft.Exchange -ConnectionUri
https://outlook.office365.com/powershell-liveid/?proxyMethod=rps -Credential $O365Cred -Authentication Basic -
AllowRedirection
Import-PSSession $O365Session
Connect-MsolService -Credential $O365Cred
```

Here is the PowerShell code:

```
Get-MailboxFolderStatistics -Identity "TargetMailBoxID" |
Select FolderType , Name , ItemsInFolder
```

Refer to the following image:

FolderType	Name	ItemsInFolder
Root	Top of Information Store	96
Calendar	Calendar	347
Contacts	Contacts	24
ImContactList	{A9E2BC46-B3A0-4243-B315-60D991004455}	0
GalContacts	GAL Contacts	0
OrganizationalContacts	Organizational Contacts	0
PeopleCentricConversationBuddies	PeopleCentricConversation Buddies	0
RecipientCache	Recipient Cache	116
QuickContacts	Skype for Business Contacts	0
ConversationActions	Conversation Action Settings	0
DeletedItems	Deleted Items	2
User Created	Backup	56
Drafts	Drafts	0
Inbox	Inbox	7127
Journal	Journal	1
JunkEmail	Junk E-mail	5
User Created	News Feed	0

We installed the EWS API 2.2 and moved the DLL file to the desired location. On the EWS managed API, we can do this using the following code:

```
#Target MailboxID's
$MailboxNames = "TargetMailID"
#Any Exchange Admin ID with appropriate permissions
$AdminID = "AdminID"
#Fetch password as secure string
$AdminPwd = Read-Host "Enter Password" -AsSecureString
#Load the Exchange Web Service DLL
$dllpath = "C:\Temp\Microsoft.Exchange.WebServices.dll"
[Reflection.Assembly]::LoadFile($dllpath)
#create a Exchange Web Service
$Service = New-Object
Microsoft.Exchange.WebServices.Data.ExchangeService([Microsoft.
Exchange.WebServices.Data.ExchangeVersion]::Exchange2013_SP1)
#Credentials to impersonate the mail box
$Service.Credentials = New-Object
System.Net.NetworkCredential($AdminID , $AdminPwd)
foreach($MailboxName in $MailboxNames)
{
#Impersonate using Exchange WebService Class
$Service.ImpersonatedUserId = New-Object
Microsoft.Exchange.WebServices.Data.ImpersonatedUserId([Microso
ft.Exchange.WebServices.Data.ConnectingIdType]::SmtpAddress,
$MailboxName)
$Service.AutodiscoverUrl($MailboxName,{$true})
```

```

#Assing EWS URL
$service.Url =
'https://outlook.office365.com/EWS/Exchange.asmx'
Write-Host "Processing Mailbox: $MailboxName" -ForegroundColor Green
#Fetch Root Folder ID
$RootFolderID = New-object Microsoft.Exchange.WebServices.Data.FolderId([Microsoft.Exchange.WebServices.Data.WellKnownFolderName]::Root, $MailboxName)
$RootFolder =
[Microsoft.Exchange.WebServices.Data.Folder]::Bind($Service,$RootFolderID)
#Create a Folder View
$FolderView = New-Object Microsoft.Exchange.WebServices.Data.FolderView(1000)
$FolderView.Traversal =
[Microsoft.Exchange.WebServices.Data.FolderTraversal]::Deep
#Retrive the Information
$response = $RootFolder.FindFolders($FolderView)
$response | Select DisplayName , TotalCount , FolderClass
}

```

This returns the same output as shown in the following image, but provides in-depth information:

DisplayName	TotalCount	FolderClass
AllContacts	267	IPF.Note
AllItems	15767	IPF
BlackBerryHandheldInfo	2	
BrokerSubscriptions	0	
Calendar Version Store	1031	IPF.Note
CalendarItemSnapshots	0	
Common Views	0	
Deferred Action	0	
Document Centric Conversations	0	IPF.Note
ExchangeSyncData	0	

You can spin up PowerShell and customize the scripting according to your needs.

At times, we may consider building a unique interface to collect and audit organization information such as AD, Mailbox, SharePoint, Lync, and so on. Let's accommodate PowerShell in C# to call the Exchange

Online cmdlets; the reason for this is the unique and clean interface, which helps us query multiple sources. In the following example, we will obtain information from Exchange Online.

Here is a demo code that will print your Exchange Online user's display name and SMTP address:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Management.Automation;
using System.Management.Automation.Remoting;
using System.Management.Automation.Runspaces;
using System.Security;
using System.Collections.ObjectModel;
using System.Text;
namespace Office365
{
    class Program
    {
        static void Main(string[] args)
        {
            string username = "ExchangeAdminID";
            string password = "Password";
            System.Security.SecureString securepassword = new
System.Security.SecureString();
            foreach (char c in password)
            {
                securepassword.AppendChar(c);
            }
            PSCredential credential = new
PSCredential(username, securepassword);
            WSManConnectionInfo connectioninfo = new
WSManConnectionInfo(new
Uri("https://ps.outlook.com/powershell"),
"http://schemas.microsoft.com/powershell/Microsoft.Exchange",
credential);
            connectioninfo.AuthenticationMechanism =
AuthenticationMechanism.Basic;
//connectioninfo.AuthenticationMechanism =
AuthenticationMechanism.Basic;
            connectioninfo.MaximumConnectionRedirectionCount =
2;
//connectioninfo.MaximumConnectionRedirectionCount
= 2;
```

```

        using (Runspace runspace =
RunspaceFactory.CreateRunspace(connectioninfo))
{
    runspace.Open();
    using (PowerShell powershell =
PowerShell.Create())
    {
        powershell.Runspace = runspace;
        //Create the command and add a parameter
        powershell.AddCommand("Get-Mailbox");

powershell.AddParameter("RecipientTypeDetails", "UserMailbox");
        //powershell.
        //Invoke the command and store the results
in a PSObject collection
        Collection<PSObject> results =
powershell.Invoke();
        foreach (PSObject result in results)
        {
            string createText =
string.Format("Name: {0} Alias: {1} Mail: {2}",
result.Properties["DisplayName"].Value.ToString(),
result.Properties["Alias"].Value.ToString(),
result.Properties["PrimarySMTPAddress"].Value.ToString());

System.IO.File.WriteAllText("C:\\User.txt", createText);
        }
    }
}
}

```

The output is illustrated in the following image:

```

file:///c:/users/cheny/documents/visual studio 2010/Projects/Office365/0
DisplayName: Anil.Yaaram Alias: Anil.Yaaram Email Anil.Yaaram@contoso.com
DisplayName: Chendrayan.Venkatesan Alias: Chendrayan.Venkatesan@contoso.com
DisplayName: Hoshank.Mahmood Alias: Hoshank.Mahmood@contoso.com
DisplayName: Karthik.Muthukumar Alias: Karthik.Muthukumar@contoso.com
DisplayName: Mostafa.Selim Alias: Mostafa.Selim@contoso.com

```

Purging items in the mailbox folder

Most organizations are moving to cloud-based storage, and as we know, all Microsoft products have a PowerShell interface. This eventually helps both developers and administrators to perform tasks without using a GUI. Not only automating tasks, using PowerShell we troubleshoot issues, identify the root cause, and so on.

Let's consider an operational requirement to purge Lync contact entries. Ignore the technical background and requirements or designs such as unified contact stores and so on; all we need to know is how to consume the EWS API and explore objects. Perform the following steps:

1. Get the target mailbox for which the items need to be purged.
2. You will need the appropriate permission to perform this task.
Connect with your exchange admin account and plan the authentication and security process based on your organization's rules.
3. Pass the admin ID credentials.
4. Load the Exchange Web Service DLL file. Run the following command:

```
#Target MailboxID's
$MailboxNames = "TargetMailBoxID1" , "TargetMailBoxID1"
#Any Exchange Admin ID with appropriate permissions
$AdminID = "AdminID"
#Fetch password as secure string
$AdminPwd = Read-Host "Enter Password" -AsSecureString
#Load the Exchange Web Service DLL
$dllpath = "C:\Temp\Microsoft.Exchange.WebServices.dll"
[Reflection.Assembly]::LoadFile($dllpath)
```

Now, perform the following steps:

1. Create an Exchange Web service.
2. Using the admin credentials, impersonate the target mailbox. Run the following command:

```
#Create a Exchange Web Service
```

```

$Service = New-Object
Microsoft.Exchange.WebServices.Data.ExchangeService([Microsoft.Exchange.WebServices.Data.ExchangeVersion]::Exchange2013_SP1)
#Credentials to impersonate the mail box
$Service.Credentials = New-Object
System.Net.NetworkCredential($AdminID , $AdminPwd)

```

After this, perform the following steps:

1. Impersonate each target mailbox using EWS.
2. Assign the EWS URL.
3. Retrieve the root folder ID.
4. Traverse and get the folder view. Here, we will use 1000 items taking the possibility of throttling into consideration.
5. Retrieve the folders from the view.
6. Query the folder name with its class.
7. Purge the items using soft delete; do not use hard delete as it makes it impossible to restore the items from the dumpster.
8. Call the `Load` method. Run the following command:

```

foreach($MailboxName in $MailboxNames)
{
#Impersonate using Exchange WebService Class
$Service.ImpersonatedUserId = New-Object
Microsoft.Exchange.WebServices.Data.ImpersonatedUserId([Microsoft.Exchange.WebServices.Data.ConnectingIdType]::SmtpAddress, $MailboxName)
$Service.AutodiscoverUrl($MailboxName, {$true})
#Assing EWS URL
$service.Url =
'https://outlook.office365.com/EWS/Exchange.asmx'
Write-Host "Processing Mailbox: $MailboxName" -
ForegroundColor Green
#Fetch Root Folder ID
$RootFolderID = New-object
Microsoft.Exchange.WebServices.Data.FolderId([Microsoft.Exchange.WebServices.Data.WellKnownFolderName]::Root,
$MailboxName)
$RootFolder =
[Microsoft.Exchange.WebServices.Data.Folder]::Bind($Service
,$RootFolderID)
#Create a Folder View

```

```

$FolderView = New-Object
Microsoft.Exchange.WebServices.Data.FolderView(1000)
$FolderView.Traversal =
[Microsoft.Exchange.WebServices.Data.FolderTraversal]::Deep
#Retrieve Folders from view
$response = $RootFolder.FindFolders($FolderView)
#Query Folder which has display name like Lync Contacts
$Folder = $response | ? {$_['FolderClass'] -eq
'IPF.Contact.MOC.QuickContacts'}
$Folder | Select DisplayName , TotalCount
#Purge the items
$Folder.Empty([Microsoft.Exchange.WebServices.Data.DeleteMode]::SoftDelete, $false)
$Folder.Load()
}

```

Deleting items from the mailbox folder

Deleting items from the mailbox folder is almost similar to purging. We can delete items using the following lines of code:

```

$Folder.Delete([Microsoft.Exchange.WebServices.Data.DeleteMode]
::SoftDelete)
$Folder.Load()

```

We cannot remove the Lync contacts entries in Outlook by simply right-clicking and deleting. We need to purge the items and then delete the entries in the contact folders. However, we can't communicate to the users to perform the delete operations themselves after performing purge, so simply executing the preceding code will complete the task. Using the previously mentioned methods we can perform administration tasks remotely without the user's intervention.

The Lync 2013 client-side API

For most of its products, Microsoft releases an SDK, using which we can automate a few things on the client side. The Microsoft Lync 2013 SDK is designed for software developers who build custom Microsoft Lync 2013 applications. It is also useful to those developers who embed the collaboration functionality in **line-of-business (LOB)** applications, which interoperate with other custom Lync SDK clients or with Lync 2013 and Microsoft Lync Server 2013.

However, this is not limited only to developers; IT professionals can also perform tasks using Windows PowerShell. In this topic, we will cover both:

- You can learn PowerShell scripting Lync 2010 SDK using the Lync Model API at <https://msdn.microsoft.com/en-us/library/office/hh243705%28v=office.14%29.aspx>
- You can learn PowerShell Scripting Lync 2010 SDK using Lync Extensibility API at <https://msdn.microsoft.com/en-us/library/gg581082.aspx>

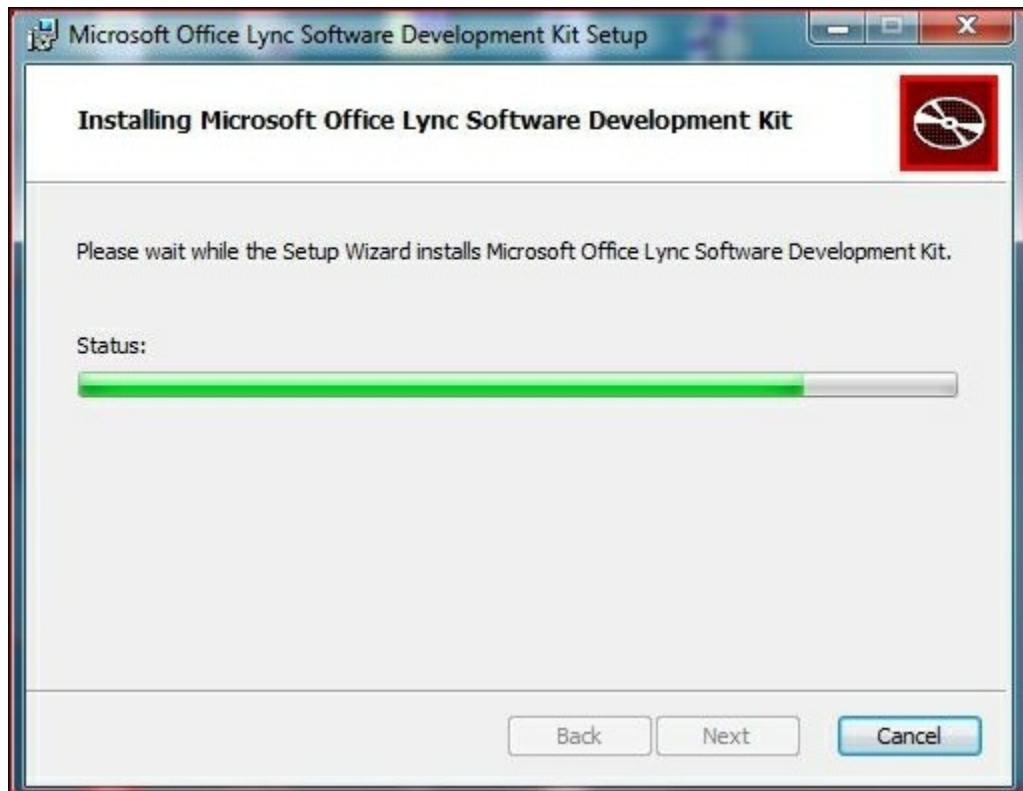
Installation of LYNC SDK

To download the LYNC client SDK refer to the following link—Read the installation documents and choose the 32 bit or 64 bit based on your LYNC client.

<https://www.microsoft.com/en-us/download/details.aspx?id=36824>

The LYNC 2013 SDK installation is just two steps as shown in the following image.

1. The first step is:



2. The second step is:



That's it, we are done with SDK Installation, now we can browse to the default location C:\Program Files (x86)\Microsoft Office\Office15\LyncSDK\Assemblies\Desktop\Microsoft.Lync.Model.dll

So, using the Import-Module cmdlet, let us load the DLL, as follows:

```
Import-Module -Name 'C:\Program Files (x86)\Microsoft Office\Office15\LyncSDK\Assemblies\Desktop\Microsoft.Lync.Model.dll'
```

Now, we will explore the LYNC client SDK in PowerShell:

```
$client = [Microsoft.Lync.Model.LyncClient]::GetClient()  
$client | GM -Force
```

The following figure is the output of the command \$client | GM | Out-GridView:

CapabilitiesChanged	Event	System.EventHandler`1[Microsoft.Lync.Model.PreferredCapabilitiesChangedEventArgs] CapabilitiesChanged(System.Object, Microsoft.Lync.Model.PreferredCapabilitiesChangedEventArgs)
ClientDisconnected	Event	System.EventHandler ClientDisconnected(System.Object, System.EventArgs)
CredentialRequested	Event	System.EventHandler`1[Microsoft.Lync.Model.CredentialRequestedEventArgs] CredentialRequested(System.Object, Microsoft.Lync.Model.CredentialRequestedEventArgs)
DelegatorClientAdded	Event	System.EventHandler`1[Microsoft.Lync.Model.DelegatorClientCollectionEventArgs] DelegatorClientAdded[System.Object, Microsoft.Lync.Model.DelegatorClientCollectionEventArgs]
DelegatorClientRemoved	Event	System.EventHandler`1[Microsoft.Lync.Model.DelegatorClientCollectionEventArgs] DelegatorClientRemoved[System.Object, Microsoft.Lync.Model.DelegatorClientCollectionEventArgs]
SignInDelayed	Event	System.EventHandler`1[Microsoft.Lync.Model.SignInDelayedEventArgs] SignInDelayed(System.Object, Microsoft.Lync.Model.SignInDelayedEventArgs)
StateChanged	Event	System.EventHandler`1[Microsoft.Lync.Model.ClientStateChangedEventArgs] StateChanged(System.Object, Microsoft.Lync.Model.ClientStateChangedEventArgs)
BeginInitialize	Method	System.IAsyncResult BeginInitialize(System.AsyncCallback callback, System.Object state)
BeginShutdown	Method	System.IAsyncResult BeginShutdown(System.AsyncCallback communicatorClientCallback, System.Object state)
BeginSignIn	Method	System.IAsyncResult BeginSignIn(string userUri, string domainAndUsername, string password, System.AsyncCallback communicatorClientCallback, System.Object state)
BeginSignOut	Method	System.IAsyncResult BeginSignOut(System.AsyncCallback communicatorClientCallback, System.Object state)
CreateApplicationRegistration	Method	Microsoft.Lync.Model.Extensibility.ApplicationRegistration CreateApplicationRegistration(string appGuid, string appName)
CreateObjRef	Method	System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
EndInitialize	Method	void EndInitialize(System.IAsyncResult asyncResult)
EndShutdown	Method	void EndShutdown(System.IAsyncResult asyncResult)
EndSignIn	Method	void EndSignIn(System.IAsyncResult asyncResult)
EndSignOut	Method	void EndSignOut(System.IAsyncResult asyncResult)
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()
ToString	Method	string ToString()
Capabilities	Property	Microsoft.Lync.Model.LyncClientCapabilityTypes Capabilities {get;}
ContactManager	Property	Microsoft.Lync.Model.ContactManager ContactManager {get;}
ConversationManager	Property	Microsoft.Lync.Model.Conversation.ConversationManager ConversationManager {get;}
DelegatorClients	Property	System.Collections.Generic.IList[Microsoft.Lync.Model.DelegatorClient] DelegatorClients {get;}

We will see a series of examples to explore the LYNC/S4B (Skype for Business) client.

Note

This works for Skype for Business clients as well. So, we will examine the functionality. If you read IM client in this topic it refers to LYNC or

Skype for Business client.

Retrieve the groups created using the following command:

```
$client = [Microsoft.Lync.Model.LyncClient]::GetClient()  
$client.ContactManager.Groups.InnerObject.Name
```

The output of this command is illustrated in the following image:

```
PS C:\windows\system32> $client = [Microsoft.Lync.Model.LyncClient]::GetClient()  
$client.ContactManager.Groups.InnerObject.Name  
Other Contacts  
Pinned Contacts  
SharePoint Portal Team  
Project Management  
End User Computing  
  
SharePoint Development
```

Execute the following command:

```
$client = [Microsoft.Lync.Model.LyncClient]::GetClient()  
$client.ContactManager | GM
```

This command returns the members of the ContactManager which is Microsoft.Lync.Model.ContactManager that has the BeginAddGroup method accepting three overloads.

```
$client = [Microsoft.Lync.Model.LyncClient]::GetClient()  
$client.ContactManager.BeginAddGroup
```

The output of this command is illustrated in the following image:

```
PS C:\windows\system32> $client = [Microsoft.Lync.Model.LyncClient]::GetClient()  
$client.ContactManager.BeginAddGroup  
  
OverloadDefinitions  
-----  
System.IAsyncResult BeginAddGroup(string customGroupName, System.AsyncCallback contactsAndGroupsCallback, System.Object state)  
System.IAsyncResult BeginAddGroup(Microsoft.Lync.Model.Group.DistributionGroup distributionGroup, System.AsyncCallback  
contactsAndGroupsCallback, System.Object state)
```

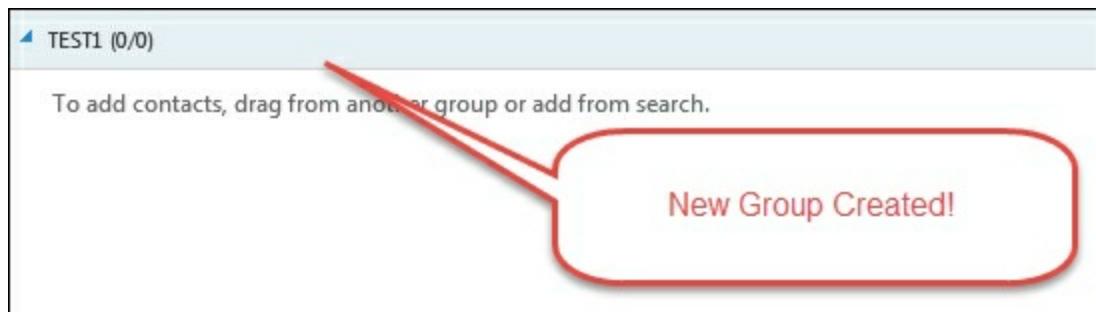
Execute the following command:

```
$client = [Microsoft.Lync.Model.LyncClient]::GetClient()  
$client.ContactManager.BeginAddGroup($GroupName, $null,$null)
```

For now, we will use the \$GroupName parameter and the \$null parameter for the other two overloads.

```
Function Create-LyncClientGroup{  
    param([Parameter(Mandatory = $true,ValueFromPipeline =  
$true)]  
        [String[]]$GroupName  
)  
    $client = [Microsoft.Lync.Model.LyncClient]::GetClient()  
    $client.ContactManager.BeginAddGroup($GroupName, $null,$null)  
}  
"Test1" | Create-LyncClientGroup
```

The above code creates a new group in the client called Test1 as shown in the following figure with no contacts:



Exploring client settings

We haven't started the extensibility API yet, so before that let us see few more client settings options. The below code shows more properties which can be viewed, as well as methods to invoke and perform tasks as shown below, even without a single click!

- Active audio devices
- SIP ID information
- Sign in configuration
- Photo information

Execute the following command to check the active audio device:

```
#To check the Active Audio Device  
$client.DeviceManager.ActiveAudioDevice
```

The output of this command is illustrated in the following image:

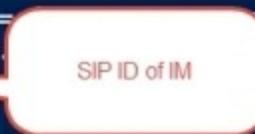
```
IsCertified : False  
Priority : 4194304  
Type : Custom  
IsActive : True  
Name : Speakers (Realtek High Definition Audio)/Microphone (Realtek High Definition Audio)  
InnerObject : Microsoft.Office.Uc.AudioDeviceClass
```

Execute the following command to check self URI:

```
#To Check Self URI  
$client.InnerObject.Self.Contact.Uri
```

The output of this command is illustrated in the following image:

```
PS C:\windows\system32> $client = [Lync.Model.LyncClient]::GetClient()  
$client.InnerObject.Self.Contact  
sip:chendrayan.venkatesan@  
PS C:\windows\system32> |
```



SIP ID of IM

Execute the following command to check sign-in configuration:

```
#To Check SigninConfiguration  
$client.SignInConfiguration
```

The output of this command is illustrated in the following image:

```

PS C:\windows\system32> $client = [Microsoft.Lync.Model.LyncClient]::GetClient()
$client.SignInConfiguration

ExternalServerUrl      : sipdir.online.lync.com:443
InternalServerUrl      : sipdir.online.lync.com:443
IsPasswordSaved        : False
Mode                   : Auto
SignedInFromIntranet   : True
SignInAvailability      : None
SignInAutoRetry         : False
TransportMode          : TcpTransport
UserName               : Chendrayan.Venkatesan@i...
InnerObject             : Microsoft.Office.Uc.SignInConfigurationClass

```



Sign in Configuration!

Execute the following command to check photo display settings:

```
#To Check if Photo Display Settings
$client.Self.PhotoDisplayed
```

This command returns True or False.

Following is the demo code to change the IM Client status:

```

Import-Module -Name 'C:\Program Files (x86)\Microsoft
Office\Office15\LyncSDK\Assemblies\Desktop\Microsoft.Lync.Model
.dll'
$Client = [Microsoft.Lync.Model.LyncClient]::GetClient()
$self = $Client.Self
$contactInfo = New-object
'System.Collections.Generic.Dictionary[Microsoft.Lync.Model.Pub
lishableContactInformationType, object]'
$contactInfo.Add([Microsoft.Lync.Model.PublishableContactInform
ationType]::Availability,6500)
$ar = $self.BeginPublishContactInformation($contactInfo, $null,
$null)
$self.EndPublishContactInformation($ar)
```

This code will change your IM client status to BUSY. 6500 is the value of BUSY. Let us make another piece of code to retrieve the values of IM client statuses—So that we can build a custom function for a quick status change without a click!

[Microsoft.Lync.Model.ContactAvailability]::Away.value

This command returns 15500.

To retrieve all values, use the following code:

```
[enum]::GetValues([System.Globalization.NumberStyles]) | %{ "  
{0,3} {1}" -f $([int]$_),$_ }
```

```
PS C:\> [enum]::GetValues([Microsoft.Lync.Model.ContactAvailability])  
None  
Free  
FreeIdle  
Busy  
BusyIdle  
DoNotDisturb  
TemporarilyAway  
Away  
Offline  
Invalid  
  
PS C:\> [enum]::GetValues([Microsoft.Lync.Model.ContactAvailability]) | %{ "{0,3} {1}" -f $([int]$_),$_ }  
0 None  
3500 Free  
6500 FreeIdle  
5000 Busy  
7500 BusyIdle  
9500 DoNotDisturb  
12500 TemporarilyAway  
15500 Away  
18500 Offline  
-1 Invalid
```

We know that PowerShell is built on .NET framework and we can consume the cmdlets in C#. We have seen a demo in the previous chapter, by applying the same concepts, let us build a module which will have just one cmdlet `Get-LyncStatus`.

In PowerShell the code is as follows:

```
$client.SigninConfiguration.SigninIntranet
```

This command returns `True` or `False`—Boolean value. In C# we will build a binary module using LYNC SDK and PowerShell.

C# code is as follows:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using System.Management.Automation;
```

```

using System.Management.Automation.Runspaces;
using Microsoft.Lync.Model;
namespace LyncAPIDemo
{
    [Cmdlet(VerbsCommon.Get, "LyncStatus")]
    public class LyncAPIDemo : PSCmdlet
    {
        protected override void ProcessRecord()
        {
            var client = LyncClient.GetClient();
            bool status =
client.SignInConfiguration.SignedInFromIntranet;
            if(status == true)
            {
                Console.WriteLine("You are signed in!");
            }
            else
            {
                Console.WriteLine("Signed Off!");
            }
        }
    }
}

```

Note

The above code will not check for existence of the LYNC client—the code assumes LYNC client/Skype for business exists.

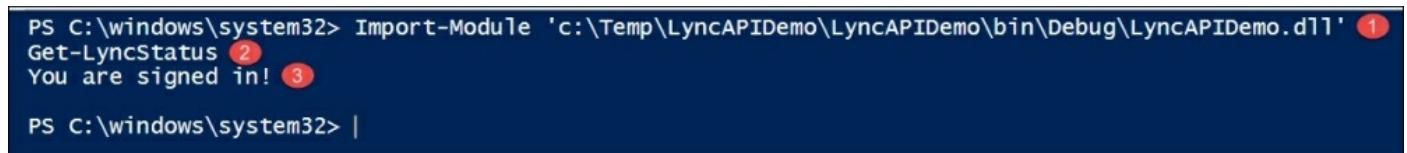
Once you have compiled the code in class library, import the module in PowerShell and make use of the cmdlet using the following command:

```

Import-Module
'c:\Temp\LyncAPIDemo\LyncAPIDemo\bin\Debug\LyncAPIDemo.dll'
Get-LyncStatus

```

The output of this command is illustrated in the following figure:



A screenshot of a Windows PowerShell window. The command 'Import-Module' is run at the prompt, followed by the path to the module file. Then, the cmdlet 'Get-LyncStatus' is run. The output shows the message 'You are signed in!'.

```

PS C:\windows\system32> Import-Module 'c:\Temp\LyncAPIDemo\LyncAPIDemo\bin\Debug\LyncAPIDemo.dll' ①
Get-LyncStatus ②
You are signed in! ③
PS C:\windows\system32>

```

Points marked in the figure are explained as follows:

- Here, we import the module
- Here, we execute the cmdlet
- Here, the details are printed—the status is true, so we got the text You are signed in!

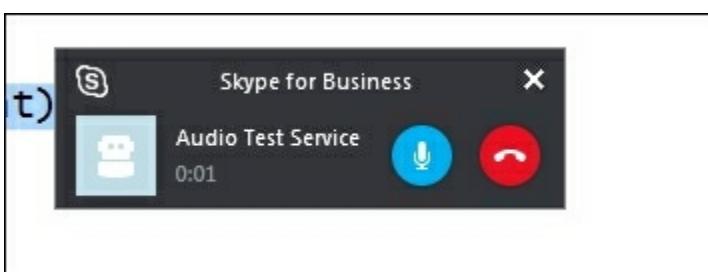
Automating test calls

Very often we may do test audio calls in Lync/Skype for Business. How about automating this? Not only to avoid clicks, but also start audio conversations with others.

The following code is very basic and just makes a test call:

```
Import-Module 'C:\Program Files (x86)\Microsoft
Office\Office15\LyncSDK\Assemblies\Desktop\Microsoft.Lync.Model
.dll'
$LyncClient = [Microsoft.Lync.Model.LyncClient]::Getclient()
$Conversation =
$LyncClient.ConversationManager.AddConversation()
[Void]$Conversation.AddParticipant($LyncClient.Self.TestCallEnd
point)
[Void]$Conversation.Modalities['AudioVideo'].BeginConnect({},0)
```

Following is the screenshot of the test call:



After the execution of this code, the audio test service begins and we need to manually disconnect. To completely automate this, the following code will help:

```
#region
Import-Module 'C:\Program Files (x86)\Microsoft
```

```

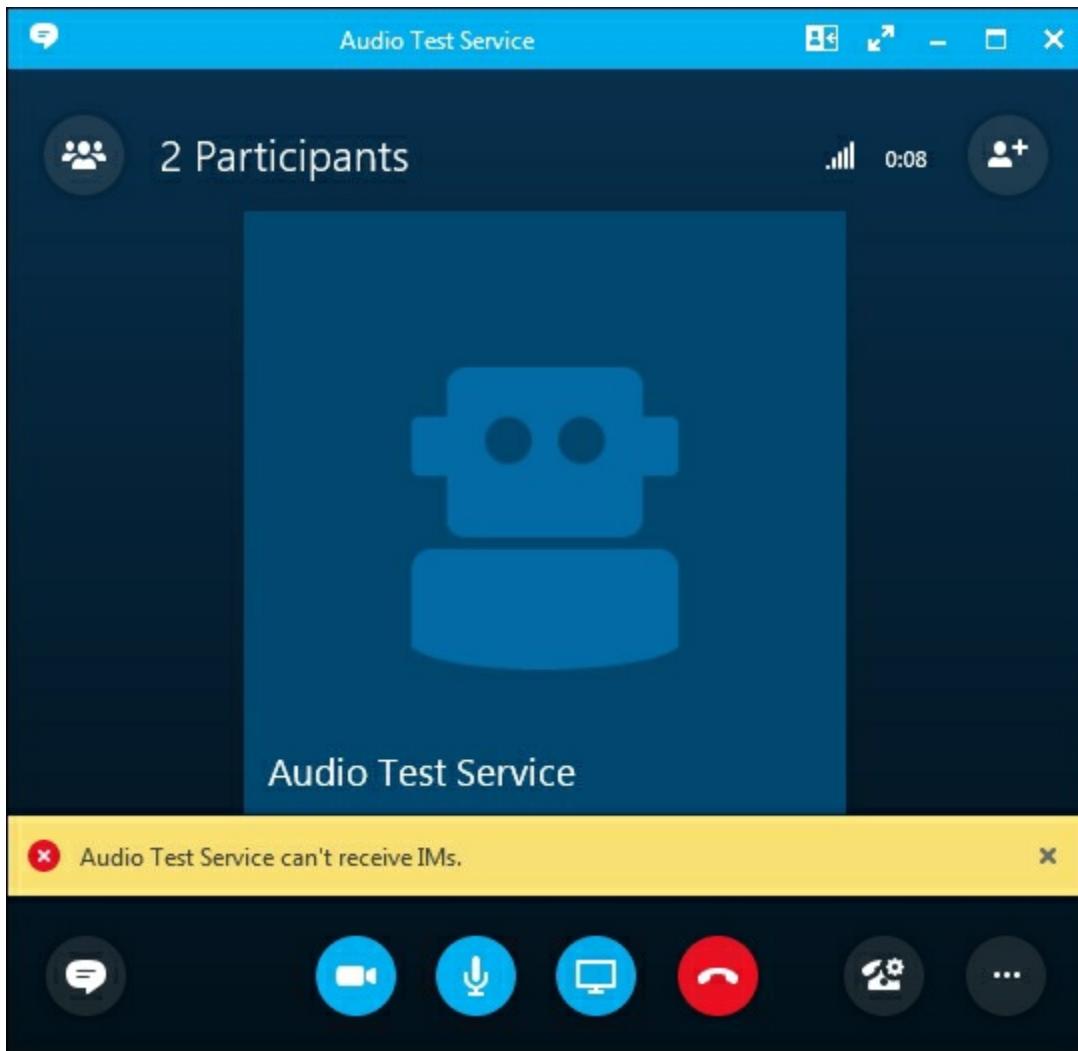
Office\Office15\SyncSDK\Assemblies\Desktop\Microsoft.Lync.Model
.dll'
$Client = [Microsoft.Lync.Model.LyncClient]::GetClient()
$TestCall = {
    $Conversation =
    $this.ConversationManager.AddConversation();

    [void]$Conversation.AddParticipant($this.Self.TestCallEndpoint),

    [void]$Conversation.Modalities['AudioVideo'].BeginConnect({},
0);
    Add-Member -InputObject $Conversation -MemberType
ScriptMethod -Name EndCall -Value {
        [void]$this.Modalities['AudioVideo'].BeginDisconnect([Microsoft
.Lync.Model.Conversation.ModalityDisconnectReason]::None, {}, 0);
    } -PassThru
}
Add-Member -InputObject $Client -MemberType ScriptMethod -Name
TestCall -Value $TestCall -Force;
#endregion
#region
$Conversation = $Client.TestCall()
Start-Sleep 15
$Conversation.EndCall()
#endregion

```

The call will disconnect after 15 seconds—during the audio test service, the IM service will not work, as shown in the following image:



IM with contacts

We can start IM conversation using LYNC SDK and PowerShell. Why? Why not use the IM client? Imagine a scenario where you need to ping a few users for testing purposes, or greet them! There are no concrete reasons for automating IM conversations, but it's good to know the flavors of .NET.

So, to do this we can use the following PowerShell code:

```
Function Send-SelfIM{
    $LyncClient =
    [Microsoft.Lync.Model.LyncClient]::GetClient()
    $ConversationManager = $LyncClient.ConversationManager;
    $Conversation = $ConversationManager.AddConversation();
```

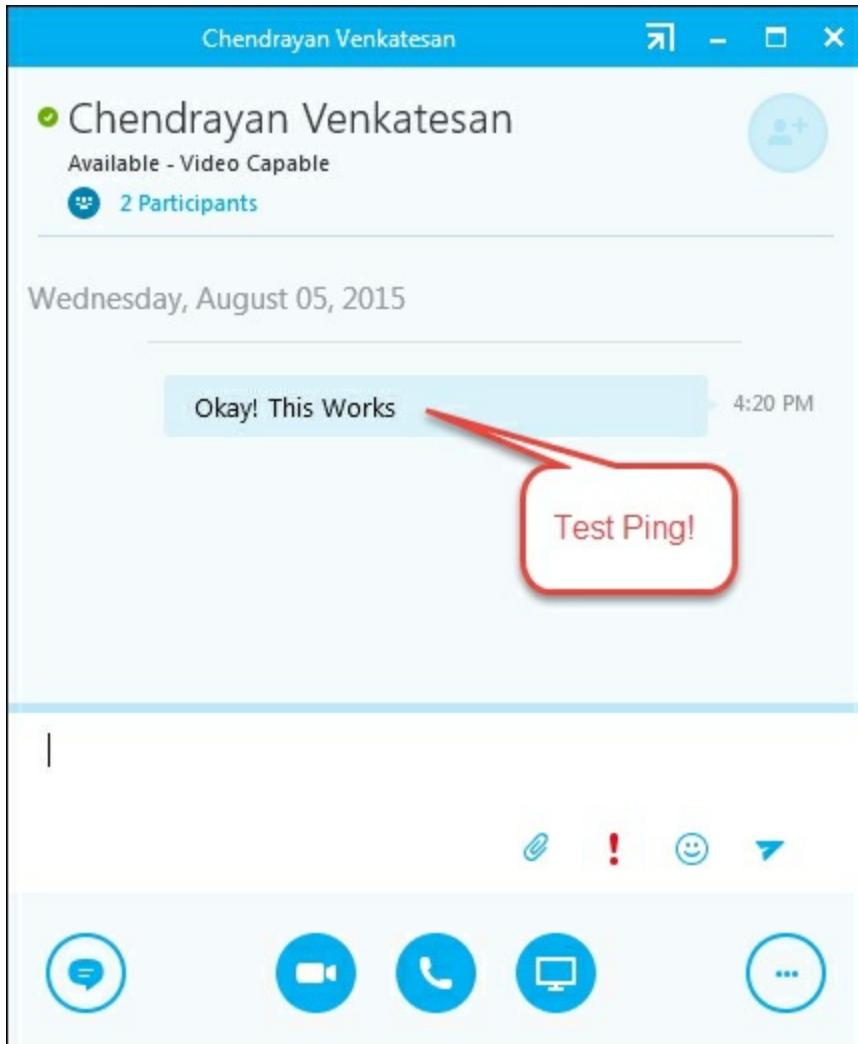
```

[void]$Conversation.AddParticipant($LyncClient.Self.Contact)

$Conversation.Modalities['InstantMessage'].BeginSendMessage("Okay! This Works", $null, $null);
}

Send-SelfIM

```



Well! You can parameterize the text link shown as follows:

```

Function Send-SelfIM{
    Param([Parameter(Mandatory = $true)]
        [String[]]$Greet
    )
    $LyncClient =
    [Microsoft.Lync.Model.LyncClient]::GetClient()
    $ConversationManager = $LyncClient.ConversationManager;
    $Conversation = $ConversationManager.AddConversation();

[void]$Conversation.AddParticipant($LyncClient.Self.Contact)

```

```
[Void]$Conversation.Modalities['InstantMessage'].BeginSendMessage($Greet,$null,$null);
}
Send-SelfIM -Greet "Hi, I am parameterized!"
```

Client-side object model – SharePoint Online

What is CSOM? What can it do for SharePoint developers and IT professionals? Most of the SharePoint developers use CSOM in the code and all are familiar with .NET. So, in this topic we would like to share CSOM and its importance. CSOM includes **JavaScript object model (JSOM)**. Developers think a lot before consuming REST, CSOM, and web services for either developing or helping IT Professionals in automation tasks. Each has its own pros and cons. It depends on when and how we use it. So, let us ignore REST and web service for now, and focus on CSOM, because this works based on the .NET Client-side object model of the SharePoint platform.

- CSOM is an acronym for Client-side object model
- CSOM provides a subset of the Server Object Model
- CSOM supports remote execution via JavaScript and .NET
- CSOM allows **Collaborative Application Markup Language (CAML)** query to query SharePoint lists
- CSOM has three distinct API, these are:
 1. .NET managed client object.

This has more subsets of functionalities which allow developers and IT professionals to make use of it

2. ECMAScript.
3. Silverlight client object model.

Following are the download links:

- SharePoint 2010 CSOM can be downloaded from
<http://www.microsoft.com/en-us/download/details.aspx?id=21786>
- SharePoint 2013 CSOM can be downloaded from
<http://www.microsoft.com/en-us/download/details.aspx?id=35585>
- SharePoint Online CSOM can be downloaded from
<http://www.microsoft.com/en-us/download/details.aspx?id=42038>

- Refer to the following CodePlex link and make use of it—please do read the documentation before you commit changes in the SharePoint environment: <http://sharepointpowershell.codeplex.com/>

The installation is similar to LYNC SDK—just download the required version and proceed with installation.

How does CSOM Work?

The basic usage of Windows PowerShell is by declaring a variable, we load the context, execute the query, and get the result from the variable. To know more about request batching using CAML, LINQ, and so on, refer to the following link:

<https://msdn.microsoft.com/en-us/library/ff798388.aspx?f=255&MSPPError=-2147217396>

In this exercise we will do different tasks on the SharePoint Online site using CSOM and PowerShell.

We have some basic information about the CSOM; this greatly integrates SharePoint IT professionals and developers.

Note

If you are looking for PowerShell tools for your Visual Studio, refer to the following link:

<https://visualstudiogallery.msdn.microsoft.com/c9eb3ba8-0c59-4944-9a62-6eee37294597>

One most common scenario in SharePoint is to audit a user profile; most of the organizations need it either to audit, or to ensure users align with corporate standards by updating their profile information. Most of the fields are mapped with active directory with a few exceptions for fields like date of birth, skills, interests, or any custom property we created in SharePoint user profile application.

In this topic we will cover a demo of below tasks using CSOM:

- Exporting user profiles
- Creating and deleting Lists
- Manipulating web settings

Using PowerShell and CSOM we can easily query the user profile information and export to CSV. Execute the following code:

```
#Import the required DLL
Import-Module 'C:\Program Files\Common Files\Microsoft
Shared\Web Server
Extensions\15\ISAPI\Microsoft.SharePoint.Client.UserProfiles.dl
l'
#Mysite URL
$site = 'https://Domain-my.sharepoint.com/'
#Admin User Principal Name
$admin = 'Admin@Domain.OnMicrosoft.Com'
#Get Password as secure String
$password = Read-Host 'Enter Password' -AsSecureString
#Get the Client Context and Bind the Site Collection
$context = New-Object
Microsoft.SharePoint.Client.ClientContext($site)
#Authenticate
$credentials = New-Object
Microsoft.SharePoint.Client.SharePointOnlineCredentials($admin
, $password)
$context.Credentials = $credentials
#Fetch the users in Site Collection
$users = $context.Web.SiteUsers
$context.Load($users)
$context.ExecuteQuery()
#Create an Object [People Manager] to retrieve profile
information
$people = New-Object
Microsoft.SharePoint.Client.UserProfiles.PeopleManager($context)

$collection = @()
Foreach($user in $users)
{
    $userprofile = $people.GetPropertiesFor($user.LoginName)
    $context.Load($userprofile)
    $context.ExecuteQuery()
    if($userprofile.Email -ne $null)
    {
```

```

$upp = $UserProfile.UserProfileProperties
foreach($ups in $upp)
{
    $profileData = "" | Select "FirstName" , "LastName"
    , "WorkEmail" , "Title" , "Responsibility"
    $profileData.FirstName = $ups.FirstName
    $profileData.LastName = $ups.LastName
    $profileData.WorkEmail = $ups.WorkEmail
    $profileData.Responsibility = $ups.'SPS-
Responsibility'
    $collection += $profileData
}
}

$collection | Export-Csv C:\Temp\SP0-UserInformation.csv -NoTypeInformation -Encoding UTF8

```

The output is illustrated in the following screenshot:

	Name	Box	B	C	D	E	F	G	H	I
1	FirstName	Lastname	WorkEmail		Title	Responsibility				
2	Chendrayan	Venkatesan	Chendrayan@domain.onmicrosoft.com		SP Admin	PowerShell SharePoint 2010				
3	Someone	Somone	HoshankMahmood@domain.onmicrosoft.com		SP Admin	PowerShell SharePoint 2010				
4										
5										
6										
7										
8										
9										
10										
11										

We can add the properties to get more information based on settings in your SharePoint farm.

Creating and deleting list

Using CSOM we can create lists and delete lists—below is the code for the same.

Following is the code for creating list:

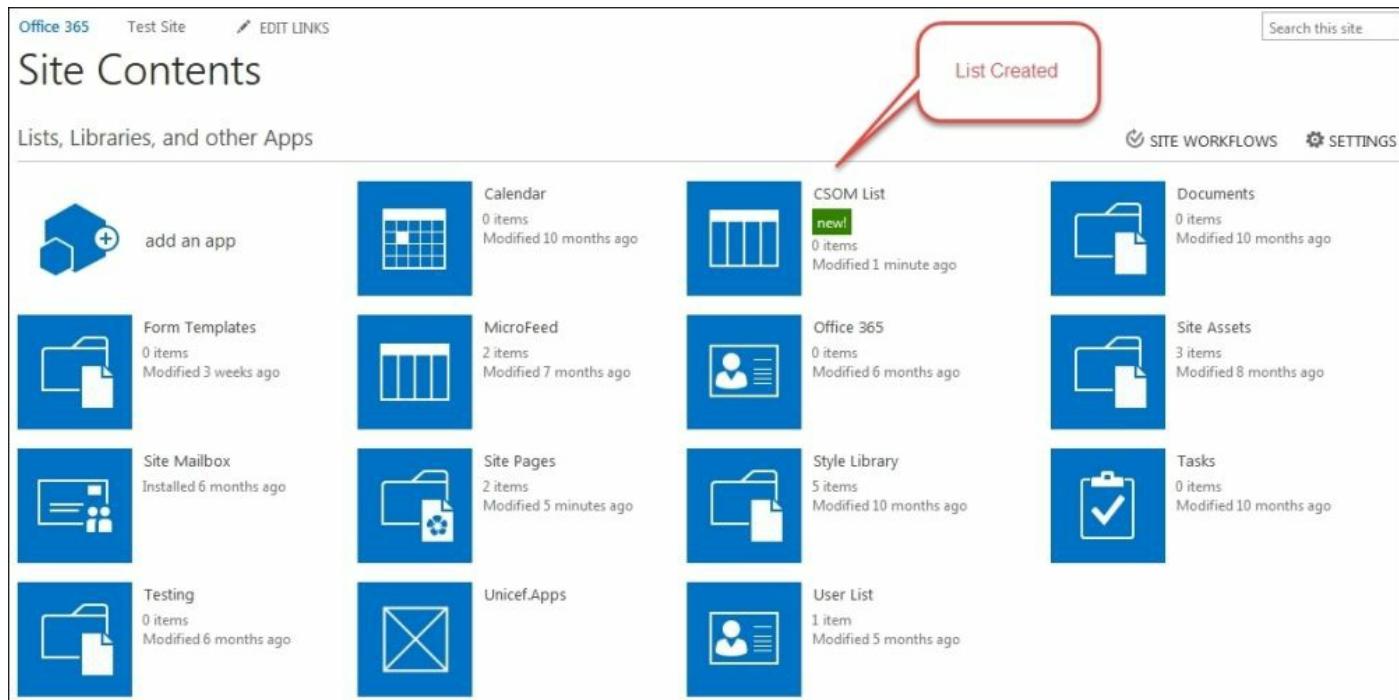
```
#Import the required DLL
```

```

Import-Module 'C:\Temp\CSOM\Microsoft.SharePoint.Client.dll'
Import-Module
'C:\Temp\CSOM\Microsoft.SharePoint.Client.Runtime.dll'
$site = 'https://Chensoffice365.sharepoint.com/'
$admin = 'Chendrayan@Chensoffice365.OnMicrosoft.Com'
$password = Read-Host 'Enter Password' -AsSecureString
$context = New-Object
Microsoft.SharePoint.Client.ClientContext($site)
$credentials = New-Object
Microsoft.SharePoint.Client.SharePointOnlineCredentials($admin
, $password)
$context.Credentials = $credentials
$site = $context.Web
$context.Load($site)
$context.ExecuteQuery()
#Create List
$listinfo =New-Object
Microsoft.SharePoint.Client.ListCreationInformation
$listinfo.Title = 'CSOM List'
$listinfo.TemplateType =
[Microsoft.SharePoint.Client.ListTemplateType]'GenericList'
$list = $Site.Lists.Add($listinfo)
$context.ExecuteQuery()
Write-Host "Successfully Created List $($listinfo.Title)"

```

After the successful execution of this code we can see the list in our site as shown in the following image:



The following code simply deletes the list. Please ensure to check the names and ID:

```
Import-Module 'C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\15\ISAPI\Microsoft.SharePoint.Client.Runtime.dll'  
#OR  
Add-Type -Path 'C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\15\ISAPI\Microsoft.SharePoint.Client.dll'  
#Site URL  
$site = 'https://Domain.sharepoint.com/'  
#Admin User Principal Name  
$admin = 'Admin@Chenoffice365.OnMicrosoft.Com'  
#Get Password as secure String  
$password = Read-Host 'Enter Password' -AsSecureString  
#Get the Client Context and Bind the Site Collection  
$context = New-Object  
Microsoft.SharePoint.Client.ClientContext($site)  
#Authenticate  
$credentials = New-Object  
Microsoft.SharePoint.Client.SharePointOnlineCredentials($admin  
, $password)  
$context.Credentials = $credentials  
#Delete List  
$list = $context.Web.Lists.GetByTitle('PowerShell CSOM')  
$context.Load($list)  
$list.DeleteObject()  
$list.Update()
```

Making PowerShell modules with SDKs

So far we have seen three different technologies, namely Exchange Online, Lync and SharePoint along with creating smaller tasks using the API with DLL and web services. In the next exercise we will build another binary module which combines all the above sample code and delivers nice PowerShell cmdlets to do the tasks. This is just a sample code you can start building on your own for your environment.

The following code is a binary module built using C# which has three cmdlets:

- Add-LyncPersonalNote: Adds/updates your LYNC personal note status
- Get-OSInformation: Retrieves OS information—just .NET way
- Get-PhotoStatus: Retrieves LYNC photo status

The code is as follows:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Management.Automation;
using Microsoft.Lync.Model;
using System.Collections.ObjectModel;
namespace OfficeServers
{
    [Cmdlet(VerbsCommon.Get,"PhotoStatus")]
    public class PhotoStatus : PSCmdlet
    {
        protected override void ProcessRecord()
        {
            //base.ProcessRecord();
            Console.WriteLine("Lync Group Name
Information....");
            var cl = LyncClient.GetClient();
            bool photo = cl.Self.PhotoDisplayed;
            if(photo == true)
            {
                Console.WriteLine("Photo will be visible to
others!");
            }
            else
            {
                Console.WriteLine("Photo is hidden!");
            }
        }
    }
    [Cmdlet(VerbsCommon.Get,"OSInformation")]
    public class OSInformation : PSCmdlet
    {
        protected override void ProcessRecord()
        {
            //base.ProcessRecord();
            Console.WriteLine("OS Version is {0}",

```

```

(Environment.OSVersion).ToString());
}
}

[Cmdlet(VerbsCommon.Add, "LyncPersonalNote")]
public class LyncPersonalNote : PSCmdlet
{
    [Parameter(Mandatory = true)]
    public string notetext{get;set;}
    protected override void ProcessRecord()
    {
        //base.ProcessRecord();
        var cl = LyncClient.GetClient();
        var self = cl.Self;
        var noteinfo = new
System.Collections.Generic.Dictionary<Microsoft.Lync.Model.Publ
ishableContactInformationType, Object>();

noteinfo.Add(Microsoft.Lync.Model.PublishableContactInformation
Type.PersonalNote,notetext);

self.BeginPublishContactInformation(noteinfo,null,null);
        //self.EndPublishContactInformation(noteinfo);
    }
}
}

```

After building the solution, the DLL will be located in the project folder. So, we need to import the module and test the output. Following are the steps:

1. Following is the command to import cmdlets:

```

Import-Module
C:\Temp\OfficeServers\OfficeServers\bin\Debug\OfficeServers
.dll -Verbose

```

This code will import the cmdlets as illustrated in the following figure:

```
Windows PowerShell
PS C:\> Import-Module C:\Temp\OfficeServers\OfficeServers\bin\Debug\OfficeServers.dll -Verbose
VERBOSE: Importing cmdlet 'Get-PhotoStatus'.
VERBOSE: Importing cmdlet 'Get-OSInformation'.
VERBOSE: Importing cmdlet 'Add-LyncPersonalNote'.
PS C:\>
```

3 cmdlets imported..
successfully!

2. Following command lists all cmdlets in the **OfficeServers** module:

```
Get-Command -Module OfficeServers
```

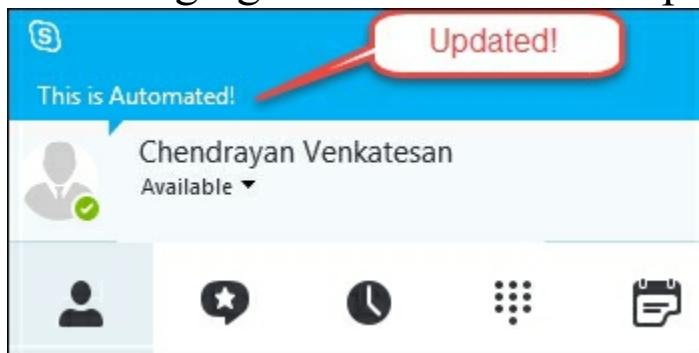
Following figure illustrates the output of this command:

```
Windows PowerShell
PS C:\> Get-Command -Module OfficeServers
 CommandType      Name          Version   Source
-----      ----          -----   -----
 Cmdlet        Add-LyncPersonalNote  1.0.0.0  OfficeServers
 Cmdlet        Get-OSInformation    1.0.0.0  OfficeServers
 Cmdlet        Get-PhotoStatus     1.0.0.0  OfficeServers
PS C:\>
```

3. Let's test the following command:

```
Add-LyncPersonalNote -notetext "This is Automated!"
```

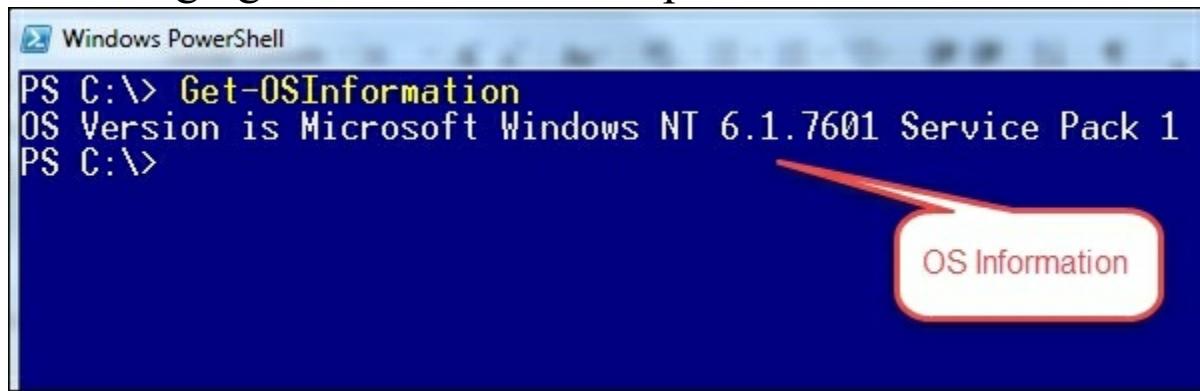
Following figure illustrates the output of this command:



Execute the following command:

```
Get-OSInformation
```

Following figure illustrates the output of this command:



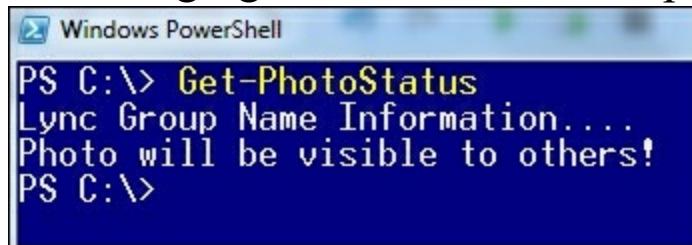
A screenshot of a Windows PowerShell window titled "Windows PowerShell". The command "Get-OSInformation" is run, and the output shows "OS Version is Microsoft Windows NT 6.1.7601 Service Pack 1". A red callout bubble points from the text "OS Version is Microsoft Windows NT 6.1.7601 Service Pack 1" to the text "OS Information" in a pink box.

```
PS C:\> Get-OSInformation
OS Version is Microsoft Windows NT 6.1.7601 Service Pack 1
PS C:\>
```

Execute the following command:

Get-PhotoStatus

Following figure illustrates the output of this command:



A screenshot of a Windows PowerShell window titled "Windows PowerShell". The command "Get-PhotoStatus" is run, and the output shows "Lync Group Name Information..." and "Photo will be visible to others!". A red callout bubble points from the text "Photo will be visible to others!" to the text "Photo Status" in a pink box.

```
PS C:\> Get-PhotoStatus
Lync Group Name Information...
Photo will be visible to others!
PS C:\>
```

Summary

In this chapter we have covered the key benefits of API and using it in Windows PowerShell. It has made IT professional's and developer's lives a lot easier than before. We know how to use API in C# and PowerShell, and making custom modules for PowerShell using C#. With reference to this you can explore more and plan for a real case scenario in your environment as well as automate tasks. So, Windows PowerShell and DSC is considered to be a one stop shop for IT pros and developers looking for swift automation, chaos free deployments, remote troubleshooting, and so on.

We have covered Exchange Web Services, LYNC client side API and SharePoint client-side object model with a walkthrough demo, creating a PowerShell module in C# for office servers.

Part 3. Module 3

Mastering Windows PowerShell Scripting

Master the art of automating and managing your Windows environment using PowerShell

Chapter 1. Variables, Arrays, and Hashes

PowerShell provides a variety of mechanisms to store, retrieve, and manipulate data used in your scripts. These storage "containers" are referred to as variables, arrays, and hashes. They can be used as containers to store strings, integers, or objects. These containers are dynamic as they automatically detect what type of data is being placed within them. Unlike other object-oriented languages, there is no need to declare the container prior to use. To declare one of these containers, you use the dollar sign (\$) and the container name.

An example of a container would look like this:

```
$myVariable
```

During this chapter, you will learn the following concepts:

- Variables
- Arrays
- Hashes
- Deciding the best container for your scripts

When you are creating names for containers, it is industry best practice to use names that are representative of the data they are storing. While containers are not case sensitive in PowerShell, it is a common practice to use *camelCase* when writing container names. *camelCase* is achieved by keeping the first letter of the container lowercase and the subsequent first letters of each word capitalized. Some variations of *camelCase* permit the first letter to be capitalized. This formatting aids in easy reading of the containers.

An example of a container using *camelCase* would look like this:

```
$webServerIPAddress
```

Variables

Variables are one of the most widely used containers in PowerShell due to their flexibility. A variable is a container that is used to store a single value or an object. Variables can contain a variety of data types including text (string), numbers (integers), or an object.

If you want to store a string, do the following:

```
$myString = "My String Has Multiple Words"  
$myString
```

The output of this is shown in the following screenshot:

```
PS C:\> $myString = "My String Has Multiple Words"  
PS C:\> $myString  
My String Has Multiple Words
```

The preceding variable will now contain the words My String Has Multiple Words. When you output the \$myString variable, as shown in the preceding screenshot, you will see that the string doesn't contain the quotations. This is because the quotations tell the PowerShell command-line interpreter to store the value that is between the two positions or quotations.

Tip

You are able to reuse variables without deleting the content already inside the variable. The PowerShell interpreter will automatically overwrite the data for you.

Subsequently, if you want to store a number, do the following:

```
$myNumber = 1  
$myNumber
```

The output of this is shown in the following screenshot:

```
PS C:\> $myNumber = 1  
PS C:\> $myNumber  
1
```

This method differentiates while storing a string as you do not use quotations. This will tell the PowerShell interpreter to always interpret the value as a number. It is important to not use quotations while using a number, as you can have errors in your script if the PowerShell interpreter mistakes a number for a string.

An example of what happens when you use strings instead of integers can be seen here:

```
$a = "1"  
$b = "2"  
$c = $a + $b  
$c
```

The output of this is shown in the following screenshot:

```
PS C:\> $a = "1"  
PS C:\> $b = "2"  
PS C:\> $c = $a + $b  
PS C:\> $c  
12
```

The `$c` variable will contain the value of 12. This is due to PowerShell interpreting your `$a` string of 1 and `$b` string of 2 and putting the characters together to make 12.

The correct method to do the math would look like this:

```
$a = 1  
$b = 2  
$c = $a + $b  
$c
```

The output of this is shown in the following screenshot:

```
PS C:\> $a = 1  
PS C:\> $b = 2  
PS C:\> $c = $a + $b  
PS C:\> $c  
3
```

Since the `$a` and `$b` variables are stored as numbers, PowerShell will perform the math on the numbers appropriately. The `$c` variable will contain the correct value of `3`.

Objects stored in variables

Objects are vastly different than strings and numbers. Objects in PowerShell are data structures that contain different attributes such as properties and methods with which one can interact. Object properties are descriptors that typically contain data about that object or other related objects. Object methods are typically sections of code that allow you to interact with that object or other objects on a system. These objects can easily be placed in variables. You can simply place an object in a variable by declaring a variable and placing an object in it. To view all of the object's attributes, you can simply call the variable containing the object, use a pipe character `|`, and use the `get-member` cmdlet.

To place an object in a variable and retrieve its attributes, you need to do this:

```
$date = get-date  
$date  
$date | get-member
```

The output is shown in the following screenshot:

```

PS C:\> $date = get-date
PS C:\> $date
Sunday, March 08, 2015 2:36:09 PM

PS C:\> $date | get-member

TypeName: System.DateTime

Name          MemberType      Definition
----          -----
Add           Method         datetime Add(timespan value)
AddDays       Method         datetime AddDays(double value)
AddHours      Method         datetime AddHours(double value)
AddMilliseconds Method        datetime AddMilliseconds(double value)
AddMinutes    Method         datetime AddMinutes(double value)
AddMonths    Method         datetime AddMonths(int months)
AddSeconds    Method         datetime AddSeconds(double value)
AddTicks      Method         datetime AddTicks(long value)
AddYears      Method         datetime AddYears(int value)
CompareTo    Method         int CompareTo(System.Object value), int
Equals        Method         bool Equals(System.Object value), bool

```

In this example, you will learn how to place an object into a variable. You first start by declaring the `$date` variable and setting it equal to the output from the `get-date` cmdlet. When you execute this, the `get-date` cmdlet references the `System.Date` class, and the `$date` variable inherits all of that object's attributes. You then call the `$date` variable and you see that the output is the date and time from when that command was run. In this instance, it is displaying the `DateTime ScriptProperty` attribute on the screen. To view all of the attributes of the `System.Date` object in the `$date` variable, you pipe those results to the `get-member` cmdlet. You will see all of the attributes of that object displayed on the screen.

If you want to use the properties and method attributes of that object, you can simply call them using dot notation. This is done by calling the variable, followed by a period, and referencing the property or method.

To reference an object's properties and method attributes, you need to do this:

```

$date = get-date
$date.Year
$date.addyears("5")

```

The output of this is shown in the following screenshot:

```
PS C:\> $date = get-date
PS C:\> $date.Year
2015
PS C:\> $date.AddYears("5")
Sunday, March 08, 2020 3:28:06 PM
```

This example shows you how to reference an object's properties and method attributes using dot notation. You first start by declaring the `$date` variable and setting it equal to the output from the `get-date` cmdlet. When you execute this, the `get-date` cmdlet references the `System.Date` class, and the `$date` variable inherits all of that object's attributes. You then leverage dot notation to reference the `Year` property attribute by calling `$date.Year`. The attribute will return `2015` as the `Year` property. You then leverage dot notation to use the `AddYears()` method to increase the years by `5`. After entering the `$date.AddYears("5")` command, you will see an output on the screen of the same month, day, and time; however, the year is incremented by `5` years.

Arrays

Arrays are the second most used containers in PowerShell. An array, in simple terms, is a multi-dimensional variable or a variable containing more than one value. The two core components to an array are the index number and the position value. When you use an array, you reference an index number and it will return the position value.

Single-dimension arrays

The following table represents an array with a single dimension:

Index number	Position value
0	Example 1
1	Example 2
2	Example 3
3	Example 4
4	Example 5

When you are storing, manipulating, or reading the data in an array, you have to reference the position in the array the data is residing. The numbers populated in the table's **Index number** column are representative of the location within the array. You will see that array's numbering starts at the number 0, and so the first data would be in cell 0. If you call the array at position 0, the result would be the position value of Example 1. When building the array, you will see that each value in the array values is separated by a comma. This tells the PowerShell interpreter to set a new array value.

First, you can start by building the array in the preceding table by entering the following command:

```
$myArray = "Example 1", "Example 2", "Example 3", "Example 4",
"Example 5"
$myArray
```

The output of this is shown in the following screenshot:

```
PS C:\> $myArray = "Example 1", "Example 2", "Example 3", "Example 4", "Example 5"
PS C:\> $myArray
Example 1
Example 2
Example 3
Example 4
Example 5
```

The preceding example displays how to create an array of strings. You first start by declaring a variable named `$myArray`. You then place multiple strings of text separated by commas to build the array. After declaring the array, you call the `$myArray` array to print the values to the console. It will return `Example 1`, `Example 2`, `Example 3`, `Example 4`, and `Example 5`.

Retrieving data at a specific position in an array is done through the use of brackets. To retrieve the value of `0` from the array, you would do the following:

```
$myArray = "Example 1", "Example 2", "Example 3", "Example 4",
"Example 5"
$myArray[0]
```

The output of this is shown in the following screenshot:

```
PS C:\> $myArray = "Example 1", "Example 2", "Example 3", "Example 4", "Example 5"
PS C:\> $myArray[0]
Example 1
```

The preceding example displays how you can obtain array data at a specific position. You first start by declaring a variable named `$myArray`. You then place multiple strings of text separated by commas to build the array. After declaring the array, you call `$myArray[0]` to access the position value of index number `0` from the array. The preceding example

returns the value of Example 1 for the index number 0.

Jagged arrays

Arrays can become more complex as you start adding dimensions. The following table represents a jagged array or an array of arrays:

Index number	Position value 0	Position value 1
0	Example 1	Red
1	Example 2	Orange
2	Example 3	Yellow
3	Example 4	Green
4	Example 5	Blue

While accessing data in a jagged array, you will need to read the cell values counting at 0 for both dimensions. When you are accessing the data, you start reading from the index number first and then the position value. For example, the Example 1 data is in the index number of 0 and the position value of 0. This would be referenced as position [0][0]. Subsequently, the data Blue is in the index number of 4 and position value of 1. This would be referenced as position [4][1].

To do this for yourself, you can build the preceding table by entering the following command:

```
$myArray = ("Example 1", "Red"), ("Example 2", "Orange"),
("Example 3", "Yellow"), ("Example 4", "Green"), ("Example 5",
"Blue")
$myArray[0][0]
$myArray[4][1]
```

The output is shown in the following screenshot:

```

PS C:\> $myArray = ("Example 1", "Red"), ("Example 2", "Orange"), ("Example 3", "Yellow"), ("Example 4", "Green"), ("Example 5", "Blue")
PS C:\> $myArray[0][0]
Example 1
PS C:\> $myArray[4][1]
Blue

```

This example displays how to create a jagged array and accessing values in the array. You first start building the jagged array by declaring the first array of "Example 1" "Red", second array of "Example 2" "Orange", third array of "Example 3" "Yellow", fourth array of "Example 4" "Green", and fifth array of "Example 5" "Blue". After building the array, you access the word Example 1 by referencing \$myArray[0][0]. You then access the word Blue by referencing \$myArray[4][1].

Updating array values

After you create an array, you may need to update the values inside the array itself. The process for updating values in an array is similar to retrieving data from the array. First you need to find the cell location that you want to update, and then you need to set that array location as equal to the new value:

Index number	Position value 0	Position value 1
0	John	Doe
1	Jane	Smith

Given the preceding table, if Jane's last name needed to be updated to display Doe instead of Smith, you would first need to locate that data record. That incorrect last name is located at index number 1 and position value 1, or [1][1]. You will then need to set that data location equal (=) to Doe.

To do this, you need to enter the following command:

```

$myArray = ("John", "Doe"), ("Jane", "Smith")
$myArray

```

```
$myArray[1][1] = "Doe"  
$myArray
```

The output of this is shown in the following screenshot:

```
PS C:\> $myArray = ("John", "Doe"), ("Jane", "Smith")  
PS C:\> $myArray  
John  
Doe  
Jane  
Smith  
PS C:\> $myArray[1][1] = "Doe"  
PS C:\> $myArray  
John  
Doe  
Jane  
Doe
```

This example displays how you can create an array and update a value in the array. You first start by defining \$myArray and use "John", "Doe", "Jane", and "Smith" as the array values. After calling the variable to print the array to the screen, you update the value in index number 1, position value 1, or \$myArray[1][1]. By setting this position equal to Doe, you change the value from Smith to Doe:

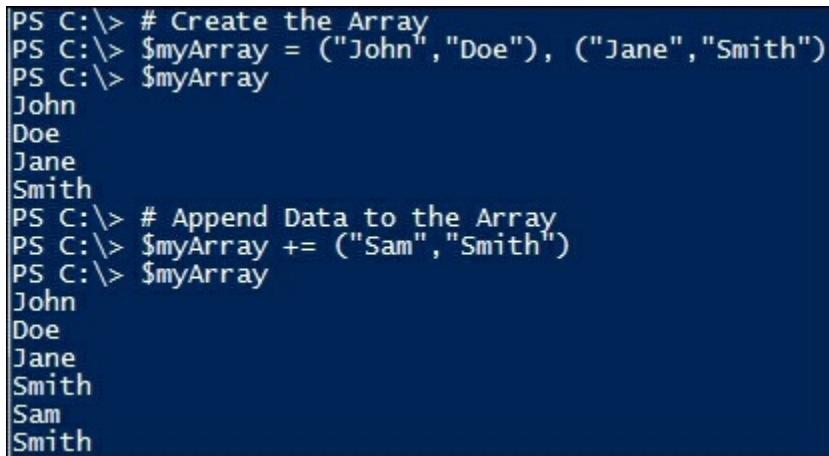
Index number	Position value 0	Position value 1
0	John	Doe
1	Jane	Smith
2	Sam	Smith

In instances where you want to append additional values to the array, you can call the array variable with the += command and the data you want to add to the array. This looks like \$array += "New Array Values". The += command is a more efficient method of performing the commands \$array = \$array + "New Array Values".

To add data into an array and make the preceding table, you can do the following operation:

```
# Create the Array
$myArray = ("John", "Doe"), ("Jane", "Smith")
$myArray
# Append Data to the Array
$myArray += ("Sam", "Smith")
$myArray
```

The output of this is shown in the following screenshot:



```
PS C:\> # Create the Array
PS C:\> $myArray = ("John", "Doe"), ("Jane", "Smith")
PS C:\> $myArray
John
Doe
Jane
Smith
PS C:\> # Append Data to the Array
PS C:\> $myArray += ("Sam", "Smith")
PS C:\> $myArray
John
Doe
Jane
Smith
Sam
Smith
```

In this example, you add values to an existing array. You first start by defining an array of \$myArray. You then print the existing contents of the array to the screen. You then add additional content by setting the array += to the new array data of ("Sam", "Smith"). After reprinting the contents of the array to the screen, you see the values Sam and Smith added to the array.

Tip

To search and remove items from an array, you will need to create a `ForEach` loop to cycle through all of the index numbers and position values. [Chapter 4, Functions, Switches, and Loop Structures](#), explores the `ForEach` looping structure.

Hashes

Hashes are used like arrays. The main difference is that they use the values as indexes versus sequentially numbered indexes. This provides easy functionality to add, remove, modify, and find data contained in the hash table. Hash tables are useful for static information that needs a direct correlation to other data:

Name	Value
John.Doe	Jdoe
Jane.Doe	jdoe1

A good example of a hash table would be in the instance of an Active Directory migration. In most Active Directory migrations, you would need to correlate old usernames to new usernames. The preceding table represents a username mapping table for these types of migrations. While a traditional array would work, a hash table makes this much easier to do.

To create the preceding hash table, enter the following command:

```
$users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
$users
```

The output of this is shown in the following screenshot:

```
PS C:\> $users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
PS C:\> $users  


| Name     | Value |
|----------|-------|
| john.doe | jdoe  |
| jane.doe | jdoe1 |


```

After you create the table, you may want to find a specific user. You can search a hash table by using the hash's indexing function. This is done by

calling `$hashName["value"]`. An example of this would look like the following command:

```
$users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
$users["john.doe"]
```

The output of this is shown in the following screenshot:

```
PS C:\> $users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
PS C:\> $users["john.doe"]  
jdoe
```

After entering the command, you will see that `$users["john.doe"]` returns `jdoe` as the correlating value in the hash.

One of the most popular methods to use with hash tables is the `add` method. The `add` method allows you to enter new values within the hash table. You can use this while building the hash table, as most hash tables are built within a script. If you want to add another user to the hash table, use the `add` method as shown here:

```
$users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
$users  
$users.add("John.Smith", "jsmith")  
$users
```

The output of this is shown in the following screenshot:

```
PS C:\> $users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
PS C:\> $users  


| Name     | Value |
|----------|-------|
| john.doe | jdoe  |
| jane.doe | jdoe1 |

  
PS C:\> $users.add("John.Smith", "jsmith")  
PS C:\> $users  


| Name       | Value  |
|------------|--------|
| jane.doe   | jdoe1  |
| john.doe   | jdoe   |
| John.Smith | jsmith |


```

You will see that `John.Smith` with the value of `jsmith` is now added to the hash table.

You can also update values in a hash by leveraging the hash's index. This is done by searching for a value and then setting its correlating hash value equal to a new value. This looks like `$arrayName["HashIndex"] = "New value"`. An example of this is given here:

```
$users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
$users  
$users["jane.doe"] = "jadoe"  
$users
```

The output of this is shown in the following screenshot:

```
PS C:\> $users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
PS C:\> $users  


| Name     | Value |
|----------|-------|
| john.doe | jdoe  |
| jane.doe | jdoe1 |

  
PS C:\> $users["jane.doe"] = "jadoe"  
PS C:\> $users  


| Name     | Value |
|----------|-------|
| jane.doe | jadoe |
| john.doe | jdoe  |


```

You will see that the mapped value for `Jane.Doe` now reads `jadoe`. This is vastly different from an array, where you would have to search for a specific value location to replace the value.

If you want to remove a user from the hash table, use the `remove` method, as shown here:

```
$users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
$users  
$users.remove("Jane.Doe")  
$users
```

The output of this is shown in the following screenshot:

```
PS C:\> $users = @{"john.doe" = "jdoe"; "jane.doe" = "jdoe1"}  
PS C:\> $users  


| Name     | Value |
|----------|-------|
| ---      | ----- |
| john.doe | jdoe  |
| jane.doe | jdoe1 |

  
PS C:\> $users.remove("Jane.Doe")  
PS C:\> $users  


| Name     | Value |
|----------|-------|
| ---      | ----- |
| john.doe | jdoe  |


```

You will see that `Jane.Doe` is now removed from the hash table. This method is helpful when you need to remove specific values, which meet certain criteria, from the hash table.

Deciding the best container for your scripts

When you are scripting, it is important to put consideration into what kind of container you will be using. Sometimes the simplicity of creating a singular variable and updating that variable is less complex than creating an array or hash table to search through. At other times, it may be more efficient to pull the whole dataset and use individual pieces of that data within your script.

Single-line variables can be used for:

- Math operations that require calculations of single or multiple values
- Catching single-line output from executing a non-PowerShell command
- Tracking current position in a loop like "percent complete"

Arrays are best used for:

- Storing a list of items for individual processing within a script
- Dumping error information from a PowerShell cmdlet

Hashes are best used for:

- Mapping data from one value to another value
- Data that requires frequent searching, updating, or building during script execution
- Storing multiple values of correlated data like user object attributes

Summary

This chapter explores the use of a variety of containers. You learned that variables, arrays, and hashes have the commonality of being able to store data, but they do it in different ways. You learned that different types of data can be stored in these containers. These types of data include numbers, strings, and objects.

This chapter explored that variables are best used for the storage of single-dimensional datasets. These datasets can contain strings but also include mathematical equations that PowerShell has the ability to inherently calculate. You also now know that arrays are primarily used in situations where you want to store more than one set of data. You are able to navigate, add, and remove values in the array based off of a starting value of 0. Last, you learned that hashes are best used while correlating data from one value to another. You are able to add, remove, and search data contained in the hash tables with the use of simple commands. In the next chapter, you will learn techniques to perform data parsing and manipulation by leveraging variables and arrays.

Chapter 2. Data Parsing and Manipulation

One of the most powerful features of PowerShell is its ability to retrieve and manipulate data. Many a times when you retrieve data from a PowerShell session, the format in which it is available is different from what you would want to display in the PowerShell window or in a log file. For this purpose, PowerShell provides powerful cmdlets and methods to perform data manipulation to best suit your needs as a PowerShell scripter.

While reading this chapter, you'll learn the following concepts:

- String manipulation
- Number manipulation and parsing
- Date/time manipulation
- Forcing data types
- PowerShell pipeline

String manipulation

String manipulation is something that you'll need to do in almost every script you create. While some of the string methods will be used more often than others, they all serve different purposes for your script. It is ultimately up to your creativity on how you want data to look when it is displayed on the screen.

To change the text to uppercase, execute the following command:

```
$a = "Error: This is an example error"  
$a.ToUpper()
```

The output of this is shown in the following screenshot:

```
PS C:\> $a = "Error: This is an example error"  
PS C:\> $a.ToUpper()  
ERROR: THIS IS AN EXAMPLE ERROR
```

The `ToUpper()` method is used to format the text to uppercase. This is helpful in situations where messages need to be emphasized or should stand out. The result of this command will change the case.

To change the string to lowercase, execute the following command:

```
$string = "The MAC Address is "
$mac = "00:A0:AA:BB:CC:DD"
$message = $string + $mac.ToLower()
$message
```

The output of this is shown in the following screenshot:

```
PS C:\> $string = "The MAC Address is "
PS C:\> $mac = "00:A0:AA:BB:CC:DD"
PS C:\> $message = $string + $mac.ToLower()
PS C:\> $message
The MAC Address is 00:a0:aa:bb:cc:dd
```

The inverse of `ToUpper()` is the use of `ToLower()`. This command will convert the entire string to lowercase in the instance when you do not want to emphasize a string. `ToLower()` is typically used in situations where a single word or a variable is uppercase, and you want to transition it to lowercase. This command shows taking two separate strings, formatting the `$MAC` string to the lowercase, and outputting both variables together.

Replacing and splitting strings

PowerShell also provides the ability to replace characters in strings using the `Replace()` method. This is useful within your scripts when you have to replace characters from the output of another method. For instance, if you pull a list of common usernames from Active Directory, they are prefixed with `cn=`. If you wanted to replace `CN=` with nothing (""), you can easily accomplish this with the `Replace()` method.

To replace items in a string, execute the following command:

```
$usernames = "CN=juser,CN=jdoe,CN=jsmith,CN=bwhite,CN=sjones"
$usernames = $usernames.replace("CN=","")
$usernames
```

The output of this is shown in the following screenshot:

```
PS C:\> $usernames = "CN=juser,CN=jdoe,CN=jsmith,CN=bwhite,CN=sjones"
PS C:\> $usernames = $usernames.replace("CN=","")
PS C:\> $usernames
juser,jdoe,jsmith,bwhite,sjones
```

This script will replace the characters `CN=` with nothing as designated by `""`. The output of this script is a list of usernames with comma separators. As you can see, this is very helpful in the manipulation of the data being output from Active Directory. Building on the prior example, if you wanted to process the usernames individually, you can leverage the `split()` method. The `split()` method will separate values in a string, by declaring a specific character to split.

To split items in a string, execute the following command:

```
$usernames = "juser,jdoe,jsmith,bwhite,sjones"
$userarray = $usernames.split(",")
$userarray
```

The output of this is shown in the following screenshot:

```
PS C:\> $usernames = "juser,jdoe,jsmith,bwhite,sjones"
PS C:\> $userarray = $usernames.split(",")
PS C:\> $userarray
juser
jdoe
jsmith
bwhite
sjones
```

When you leverage the `split()` method, as shown in this example, the script uses the comma as designation that the next item needs to be a new value within the array. The output from this script allows you to interact with these usernames individually. You will frequently use the `split()` method while working with **comma separated values (CSV)**.

files or XML files. It's common for these types of files to contain multiple objects per line, which makes sense to leverage the `split()` method.

Counting and trimming strings

PowerShell has two methods to count objects within variables and arrays. The first is done by using the `Count` method. The `Count` method is used to count the number of objects that are contained within an array. This is useful when you are attempting to determine the quantity of items you'll be processing within your script.

To count items in an array, execute the following command:

```
$services = get-service  
$services.count
```

The output of this is shown in the following screenshot:



```
PS C:\> $services = get-service  
PS C:\> $services.count  
187
```

A good example of determining how many objects are present in an array is obtaining the number of services running on a system. The previous command displays a query of the services on a box and uses the `Count` method to obtain the quantity of services. The system in this example has 210 services.

The second method to count objects is used for instances in your scripts where you need to determine the length or the number of characters in a string. This is completed by using the `Length` method. The `Length` method will count the number of characters in a string, including spaces, and output the quantity of characters.

A common scenario where you would use the `Length` method is with Windows file and folder paths. Since the Windows operating systems are

well known for having a file path limitation of 255 characters, we can leverage the `Length` method to qualify in cases where the path is over that limitation.

To get the length of a string, execute the following command:

```
$path = "c:\windows\system32\drivers\1394bus.sys"  
$path.length
```

The output of this is shown in the following screenshot:

```
PS C:\> $path = "c:\windows\system32\drivers\1394bus.sys"  
PS C:\> $path.length  
39
```

In this example, you are counting the number of characters in a file and folder path to ensure that you do not exceed the maximum number of characters. This command counted the length of `$path` to be 39 characters. Through this command, you've determined that it is not over the limitations of the Windows operating system.

The Trim method

As you script, you'll run into situations where the output from a command or the input from a file may not be in a format that can be easily parsed. For example, if you import values from a CSV file and those values have extra spaces, it can cause your script to fail. PowerShell provides the ability to trim strings of spaces and other characters with the `Trim()` method. The `Trim()` method comes in three different variations, which are `Trim()`, `TrimStart()`, and `TrimEnd()`.

To trim the spaces out of a string, use the following command:

```
$csvValue = "    servername.mydomain.com    "  
$csvValue.Trim()
```

The output of this is shown in the following screenshot:

```
PS C:\> $csvValue = " servername.mydomain.com "
PS C:\> $csvValue.trim()
servername.mydomain.com
```

By default, when you use the `Trim()` method without declaring any characters to trim, it will automatically remove the spaces that surround the text values in a string. This example displays a computer name that is surrounded by spaces. After running the `Trim()` method, you'll see that the spaces are successfully trimmed from the string.

To trim values out of a string, execute the following command:

```
$csvValue = "servername.mydomain.com"
$csvValue = $csvValue.trim(".mydomain.com")
$csvValue
```

The output of this is shown in the following screenshot:

```
PS C:\> $csvValue = "servername.mydomain.com"
PS C:\> $csvValue = $csvValue.trim(".mydomain.com")
PS C:\> $csvValue
servername
```

If you want to trim a specific value from a string, you can declare it within the `Trim()` method. By executing the preceding script, you'll see that declaring a text value of `.mydomain.com` within the `Trim()` method will remove those characters from the entire string.

The `TrimStart()` method provides the same functionality of the `Trim()` method; however, it only removes characters from the beginning of the string. Likewise, the `TrimEnd()` method will only remove characters from the ending of the string. These are helpful in situations where you need to parse the data into values that can be read easily.

To trim the beginning and end of a string, execute the following command:

```
$csvValue = "FQDN: servername.mydomain.com"
```

```
$csvValue = $csvValue.trimStart("FQDN: ")
$csvValue = $csvValue.trimEnd(".mydomain.com")
$csvValue
```

The output of this is shown in the following screenshot:

```
PS C:\> $csvValue = "FQDN: servername.mydomain.com"
PS C:\> $csvValue = $csvValue.trimStart("FQDN: ")
PS C:\> $csvValue = $csvValue.trimEnd(".mydomain.com")
PS C:\> $csvValue
servername
```

The preceding example displays the use of both `TrimStart()` and `TrimEnd()` methods. In this example, you trim the "FQDN: " characters from the start of the string and `.mydomain.com` from the end of the string. The final output of this is `servername`.

When you use the `Trim()` method, you'll want to remember that it will remove all instances of the words found at the beginning and ending of the string. If you used `computername` instead of `servername`, you would have noticed that the output from the method would have `putername`. The `Trim()` method would have matched the words `com` and removed it from the string. It is best to use the `Trim()` method to remove spaces and unneeded characters in strings. Use the `Replace()` method to remove series of strings such as `.mydomain.com`.

This would look like this:

```
$csvValue = "computername.mydomain.com"
$csvValue.Trim("com")
```

The output of this is shown in the following screenshot:

```
PS C:\> $csvValue = "computername.mydomain.com"
PS C:\> $csvValue.Trim("com")
putername.mydomain.
```

The preceding example displays how the `Trim()` method will remove strings from the front and the end of a string. You first need to start by

declaring the `$csvValue` variable as equal to `computername.mydomain.com`. You then need to leverage the `Trim()` method on the variable searching for the word `com`. You'll see that the method trims both the beginning `com` and end `com` on the string itself. The end result is `putername.mydomain`.

The Substring method

The `Substring()` method is another string manipulator within the PowerShell toolset. It is based on the requirement that you may want to remove characters present at a fixed position within a string. The following table displays the string positions for the string `TESTING123`. Like an array, the string position starts counting at 0, as shown here:

String position	0	1	2	3	4	5	6	7	8	9
String value	T	E	S	T	I	N	G	1	2	3

To obtain a substring from a string, execute the following command:

```
$string = "TESTING123"  
$string = $string.substring("7")  
$string
```

The output of this is shown in the following screenshot:

```
PS C:\> $string = "TESTING123"  
PS C:\> $string = $string.substring("7")  
PS C:\> $string  
123
```

The `Substring()` method is designed to extract data present at specific locations in a string. If you wanted to extract the numbers `123` from the preceding table, you can use the `Substring()` method referencing the start position of 7. All of the remaining characters after position 7 will be displayed and the output of the method is `123`.

To obtain a substring range from a string, execute the following

command:

```
$string = "TESTING123"  
$string = $string.substring("0","4")  
$string
```

The output of this is shown in the following screenshot:

```
PS C:\> $string = "TESTING123"  
PS C:\> $string = $string.substring("0","4")  
PS C:\> $string  
TEST
```

The `Substring()` method allows you to enter a second value within the method. While the first value designates the start position, the second value designates how many characters after the start position you want to include. In the previous script example, the script starts at position 0 and counts 4 spaces after the position of 0. The result of this command is TEST.

The string true and false methods

PowerShell has built-in string searching capabilities that provide you with the ability to quickly determine whether a string contains a specific value. The three methods that can perform the searching in a string are `Contains()`, `Startswith()`, and `Endswith()`. All of these methods are based on the same principle, that is, finding a specific value and reporting `True` or `False`.

To see whether a string contains a value, do this:

```
$ping = ping ThisDoesNotExistTesting.com -r 1  
$ping  
$deadlink = $ping.contains("Ping request could not find host")  
$deadlink
```

The output of this is shown in the following screenshot:

```
PS C:\> $ping = ping ThisDoesNotExistTesting.com -r 1
PS C:\> $ping
Ping request could not find host ThisDoesNotExistTesting.com. Please check the name and try again.
PS C:\> $deadlink = $ping.Contains("Ping request could not find host")
PS C:\> $deadlink
True
```

This example leverages the ping command to determine whether a specific website or host is alive. In our example, you capture the ping command in the \$ping variable. You then search that variable for text that matches Ping request could not find host. As the output from the ping command returns the value you are looking for, the Contains() method will return True and the \$deadlink variable is set to True.

To see whether a string starts with a value, execute the following command:

```
$ping = ping ThisDoesNotExistTesting.com -r 1
$ping
$deadlink = $ping.StartsWith("Ping request could not find
host")
$deadlink
```

The output of this is shown in the following screenshot:

```
PS C:\> $ping = ping ThisDoesNotExistTesting.com -r 1
PS C:\> $ping
Ping request could not find host ThisDoesNotExistTesting.com. Please check the name and try again.
PS C:\> $deadlink = $ping.StartsWith("Ping request could not find host")
PS C:\> $deadlink
True
```

When you run the same script with the Startswith() method, it will return the same result of True. That is because the value that you are searching for starts with Ping request could not find host.

To see whether a string ends with a specific value, execute the following command:

```
$ping = ping ThisDoesNotExistTesting.com -r 1
$ping
$deadlink = $ping.EndsWith("Please check the name and try
```

```
again.")  
$deadlink
```

The output of this is shown in the following screenshot:

```
PS C:\> $ping = ping ThisDoesNotExistTesting.com -r 1  
PS C:\> $ping  
Ping request could not find host ThisDoesNotExistTesting.com. Please check the name and try again.  
PS C:\> $deadlink = $ping.EndsWith("Please check the name and try again.")  
PS C:\> $deadlink  
True
```

When you run a similar script with the `EndsWith()` method, it will return the result of `True`. That is because the value that you are searching for ends with `Please check the name and try again.`.

Number manipulation and parsing

PowerShell is a powerful mathematics calculator. In fact, PowerShell has an entire Windows class dedicated to mathematics calculations that can be called by using the `[System.Math]` .NET class. When you are working with the `[System.Math]` classes, it is common to call *static fields* within a class. Static fields are static properties, methods, and objects that can be called to display data or do actions. To call a static field, you call the `[Math]` (shortened version of `[System.Math]`) class, followed by two colons `::` and the static field name.

To use the math operation to calculate pi, execute the following command:

```
[math]::pi
```

The output of this is shown in the following screenshot:

```
PS C:\> [math]::pi  
3.14159265358979
```

This simple command will provide *PI* if you ever need it for a calculation by using the `pi` method of the `math` class. The result of this command returns `3.14159265358979`.

To use the math operation to calculate Euler's number, execute the following command:

```
[math]::e
```

The output of this is shown in the following screenshot:

```
PS C:\> [math]::e  
2.71828182845905
```

Likewise, if you ever need to reference Euler's Number (e), you can achieve this by leveraging the `e` method of the `math` class. The result of this command returns `2.71828182845905`.

To calculate the square root of a number, execute the following command:

```
[math] ::sqrt("996004")
```

The output of this is shown in the following screenshot:

```
PS C:\> [math]::sqrt("996004")
998
```

If you need to calculate the square root of a large number, you can use the `sqrt` method of the `math` class. The result of this command returns `998`.

To round a number, execute the following command:

```
$number = "214.123857123495731234948327312341657"
[math] ::Round($number,"5")
```

The output of this is shown in the following screenshot:

```
PS C:\> $number = "214.123857123495731234948327312341657"
PS C:\> [math] ::Round($number,"5")
214.12386
```

Rounding is very common for integers in your scripts. When you want to use the `Round()` method, you'll have to specify a number and the number of digits you want to round it to. In this command, you take the number of `214.123857123495731234948327312341657` and round it to the fifth digit. The result of this command returns `214.12386`.

Formatting numbers

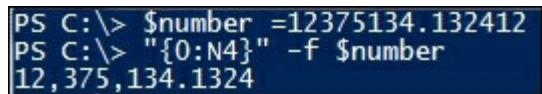
While PowerShell can perform mathematics very well, it does not have any native commands for the formatting of numbers. In order to format numbers, you'll need to leverage PowerShell's ability to use the .NET Framework's formatting methods. The construct for the .NET Framework format methods is called by specifying "

{StartingCharacter:FormatTypePrecision}" -f \$variable. The start character is the position where you want to start formatting the number. The most common format types are currency (c), decimal (d), numeric (n), percentage (p), and hexadecimal (x). The precision field is the number of decimal places you want the number to be accurate to.

To format your number in a numeric formatting, execute the following command:

```
$number =12375134.132412  
"{0:N4}" -f $number
```

The output of this is shown in the following screenshot:

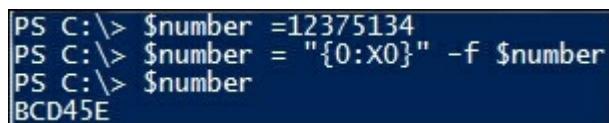
A screenshot of a Windows PowerShell window. The command entered is "\$number =12375134.132412" followed by "{0:N4}" -f \$number. The output displayed is "12,375,134.1324".

In this example, you are taking the 12375134.132412 number and formatting it in numeric format starting at the first character. This command also rounds the number to the fourth digit.

To format a number to make it hexadecimal, execute the following command:

```
$number =12375134  
$number = "{0:X0}" -f $number  
$number
```

The output of this is shown in the following screenshot:

A screenshot of a Windows PowerShell window. The command entered is "\$number =12375134" followed by "\$number = "{0:X0}" -f \$number". The output displayed is "BCD45E".

If you want to convert an integer to hexadecimal, you can format the integer to hexadecimal by specifying "`{0:X0}`" `-f $number`. The output of this command is `BCD45E`. It is important to remember that hexadecimal only supports formatting from whole numbers. If you do not use whole numbers, the script will fail due to it being in an invalid format.

Formatting bytes

PowerShell has the ability to directly convert numbers to kilobytes (`KB`), megabytes (`MB`), gigabytes (`GB`), and terabytes (`TB`) through predefined aliases for conversions. This is helpful when you are pulling data values, which by default, are formatted in bytes. Some of these may include disk space and memory on a system. The use of the alias is number in bytes divided by one unit of measure.

To format bytes to KB, MB, GB, and TB, use the following operations:

```
# 16 GB of Memory in Bytes
$ComputerMemory = 16849174528
$ComputerMemory / 1TB
$ComputerMemory / 1GB
$ComputerMemory / 1MB
$ComputerMemory / 1KB
```

The output of this is shown in the following screenshot:

```
PS C:\> # 16 GB of Memory in Bytes
PS C:\> $ComputerMemory = 16849174528
PS C:\> $ComputerMemory / 1TB
0.0153242349624634
PS C:\> $ComputerMemory / 1GB
15.6920166015625
PS C:\> $ComputerMemory / 1MB
16068.625
PS C:\> $ComputerMemory / 1KB
16454272
```

When you execute the preceding script, PowerShell will take the memory size of a computer in bytes and convert it to terabytes, gigabytes, megabytes, and kilobytes. As you can see, PowerShell

provides a quick ability to determine data calculations using the predefined aliases.

Date and time manipulation

When you are scripting, there are times where you may need to get the date and time of a system. PowerShell offers the `get-date` cmdlet, which provides the date and time in many different formats of your choice.

To obtain the date object, execute the following command:

```
$time = get-date  
$time
```

The output of this is shown in the following screenshot:

```
PS C:\> $time = get-date  
PS C:\> $time  
Tuesday, March 10, 2015 4:51:49 PM
```

The standard `get-date` cmdlet, without any triggers, will generate the long date and time format. When you store the `date` object in a variable, it is important to remember that the data captured from the cmdlet is a snapshot in time. You'll have to call the `get-date` cmdlet again to get new values for the updated date and time.

The following table displays all of the date time formatting codes:

Format code	Result	Example
MM	Month in numeric format	04
DD	Day in numeric format	15
YYYY	Year in numeric format	2014
HH	Hour in numeric format (24hrs)	14

hh	Hour in numeric format (12hrs)	02
mm	Minutes in numeric format	15
ss	Seconds in numeric format	12
tt	AM/PM (12hr)	PM

When you call the `get-date` command, you also have the ability to format it in multiple ways using the `-format` property. The preceding table displays different formatting options you can use to create your own date time format. These values are case-sensitive.

To format the date object to specific values, execute the following command:

```
$date = get-date -format "MM/dd/yyyy HH:MM:ss tt"
$date
```

The output of this is shown in the following screenshot:

```
PS C:\> $date = get-date -format "MM/dd/yyyy HH:MM:ss tt"
PS C:\> $date
03/10/2015 16:03:31 PM
```

The previous command displays how you leverage the `get-date` cmdlet with the `-format` trigger. When you execute the command, it returns the values for the month, day, year, hours, minutes, seconds, and the AM/PM indicator. As you can see, you can leverage the date time formatting in conjunction with strings and other characters to create the time format you desire.

To format the date object and insert it between strings, you can execute the following command:

```
$date = get-date -format "MMddyyyyHHMMss"
$logfile = "Script" + $date + ".log"
$logfile
```

The output of this is shown in the following screenshot:

```
PS C:\> $date = get-date -format "MMddyyyyHHMMss"  
PS C:\> $logfile = "Script" + $date + ".log"  
PS C:\> $logfile  
Script03102015160327.log
```

The preceding example displays how you can leverage the `get-date` cmdlet, with the `-format` trigger to generate a name for a log file. This is helpful in situations where you may have to run a command in PowerShell frequently, and you have to label the execution time. The preceding script will generate the date and time, append the word `Script` in front, and `.log` at the end of the string. The resulting filename from this is unique.

To add days using the `date` object, execute the following command:

```
$date = (get-date).AddDays(30).ToString("MM/dd/yyyy")  
$date
```

The output of this is shown in the following screenshot:

```
PS C:\> $date = (get-date).AddDays(30).ToString("MM/dd/yyyy")  
PS C:\> $date  
04/09/2015
```

The `get-date` cmdlet is also robust enough to be able to perform math operations with dates. The preceding example will take the current date, add 30 days to it, and set it to the `$date` variable. You can also use `AddYears()`, `AddMonths()`, `AddHours()`, `AddSeconds()`, `AddMilliseconds()`, and `AddTicks()` to increase the time. If you want to use subtraction, you can enter a negative value in the method and it will subtract that value from the methods. This would look like `AddDays(-30)` to subtract 30 days.

Note

For more information on date time formatting values, you can go to

<http://technet.microsoft.com/en-us/library/ee692801.aspx>.

The last formatting technique important for scripting is converting system time/ticks to legible time formats. This is achieved by calling the [DateTime] class and leveraging the `FromFileTime` method. The preceding example displays formatting the tick number of 130752344000000000, leveraging the [DateTime] class, and formatting it to Monday, May 04, 2015 1:33:20 PM. This is useful for system attributes that are only displayed in tick format such as `LastLogonTimestamp` or `LastBootUpTime`.

To convert file time to a different format, execute the following command:

```
$date = [datetime]::FromFileTime("130752344000000000")
$date
```

The output of this is shown in the following screenshot:

```
PS C:\> $date = [datetime]::FromFileTime("130752344000000000")
PS C:\> $date
Monday, May 04, 2015 12:33:20 PM
```

The preceding example displays how to take system ticks and convert them into a legible date time format. You first start by declaring a `$date` variable. You then call the `[datetime]` class and reference the `FromFileTime` static field. You feed the tick time of 130752344000000000 into the static field. This formats the tick time to the default date time format and stores the value in the `$date` variable. You then call the `$date` variable, and you see the converted value of Monday, May 04, 2015 1:33:20 PM.

Forcing data types

While developing scripts, you may run into instances where you may want to force a specific data type. This is helpful in cases where PowerShell automatically interprets the output from a command incorrectly. You can force data types by the use of brackets specifying a data type and a variable.

To force a string data type, execute the following command:

```
[string]$myString = "Forcing a String Container"  
$myString
```

The output of this is shown in the following screenshot:

```
PS C:\> [string]$myString = "Forcing a String Container"  
PS C:\> $myString  
Forcing a String Container
```

The preceding command forces the string data type to the \$myString variable. The result is that the \$myString variable will always remain a string. It is important to know that if the object or item that you are trying to force to a data type doesn't have a direct conversion to that data type, it will throw an error or exception. This would be the case if you try to insert a string into an integer data type.

To force a string data type and generate a data exception, execute the following command:

```
[int]$myInt = "Trying to Place a String in an Int Container"
```

The output of this is shown in the following screenshot:

```
PS C:\> [int]$myInt = "Trying to Place a String in an Int Container"  
Cannot convert value "Trying to Place a String in an Int Container" to type "System.Int32". Error: "Input string was  
not in a correct format."  
At line:1 char:1  
+ [int]$myInt = "Trying to Place a String in an Int Container"  
+ ~~~~~  
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException  
+ FullyQualifiedErrorId : RuntimeException
```

The preceding example displays trying to insert a string into an [int] data type. You first start by forcing the \$myInt variable to be a [int] data type. You then try to set that equal to a string value of "Trying to Place a String in an Int Container". After entering the command, you immediately receive an exception of Cannot convert value "Trying to Place a String in an Int Container" to type System.Int32. This example shows that you cannot mix and match data types that do not have direct conversions to each other.

There are a variety of data types that you can force within PowerShell. The following table represents the common data types for use with PowerShell and an explanation and an example of its use:

Data type	Explanation	Example	\$a value
[string]	String of Unicode characters	[string]\$a = "Hello"	Hello
[char]	A Unicode 16-bit character	[char]\$a = 0xA9	©
[byte]	An 8-bit character	[byte]\$a = 0x0001D	29
[int]	32-bit integer	[int]\$a = 12345	12345
[long]	64-bit integer	[long]\$a = 1234.243	1234
[bool]	Boolean True/False value	[bool]\$a = 1	True
[decimal]	A 128-bit decimal	[decimal]\$a = 1234.243	1234.243
[single]	A single-precision 32-bit number	[single]\$a = 1234.243	1234.243
[double]	A double-precision 64-bit number	[double]\$a = 1234.243	1234.243

[datetime]	A data time value	[datetime]\$a = "01-APR-2014"	Tuesday, April 1, 2014 12:00:00 AM
[xml]	A XML-styled value	[xml]\$a = "<test> <a>Testing</test>" \$a.test.a	Testing
[array]	An array-styled value	[array]\$a = 1,2,3	1 2 3
[hashtable]	A hashtable-styled value	[hashtable]\$a = @{"Old" = "New"} ----- Old	Name ----- Old Value ----- New

Piping variables

The concept of piping isn't anything new to the scripting world. Piping, by definition, is directing the output of an object to another object. When you use piping in PowerShell, you are taking the output of one command and sending the data for use with another section of code. The manipulation can be either to a more legible format, or can be by selecting a specific object and digging deeper into those attributes. A pipe is designated by the '| ' symbol and is used after you enter a command. The construct of a pipe looks like this: `command | ResultManipulation | SortingObjects`. If you need to access the individual items in the pipeline, you can leverage the pipeline output `$_` command. This tells the pipeline to evaluate the results from the pipeline and their attributes.

The pipeline offers a wide variety of uses; you can leverage commands such as `sort-object` to sort by a specific attribute, `format-list` to format the objects into a list, and even the `select-object` where you can select specific attributes to form the pipeline for additional processing. `Select-object` also allows you to leverage the `-first` and `-last` parameters with a number to select a record set from the beginning or ending of the pipeline. Another popular command is the `where` command, which allows you to write an expression to select items in the pipeline that meet certain criteria.

To pipe values from a cmdlet, execute the following command:

```
$services = get-service | where{$_.name -like "*Event*"} | Sort-object name  
$services
```

The output of this is shown in the following screenshot:

```
PS C:\> $services = get-service | where{$_.name -like "*Event*"} | Sort-object name  
PS C:\> $services  


| Status  | Name        | DisplayName       |
|---------|-------------|-------------------|
| Running | eventlog    | Windows Event Log |
| Running | EventSystem | COM+ Event System |


```

This example displays the use of piping the `get-services` cmdlet. It starts by getting all the services on a system, the user then *pipes* those results to the selection criteria where the object's name is like the word `event`, which then *pipes* those results to the sorting of objects in alphabetical order by their name property. The output is only the services with the names that contain `Event` in alphabetical order.

To obtain all files that are larger than a specific size, execute the following command:

```
$largeFiles = get-childitem "c:\windows\system32\" |  
where{$_.length -gt 20MB}  
$count = $largeFiles.count  
Write-host "There are $count Files over 20MB"  
write-host "Files Over 20MB in c:\Windows\System32\ :"  
$largefiles | select-object name,length,lastwritetime | format-list
```

The output of this is shown in the following screenshot:

```
PS C:\> $largeFiles = get-childitem "c:\windows\system32\" | where{$_.length -gt 20MB}  
PS C:\> $count = $largeFiles.count  
PS C:\> write-host "There are $count Files over 20MB"  
There are 4 Files over 20MB  
PS C:\> write-host "Files Over 20MB in c:\Windows\System32\ :"  
Files Over 20MB in c:\Windows\System32\ :  
PS C:\> $largefiles | select-object name,length,lastwritetime | format-list  
  
Name      : igdfcl64.dll  
Length    : 23048704  
LastWriteTime : 05/21/2014 12:33:36 AM  
  
Name      : MRT.exe  
Length    : 86054176  
LastWriteTime : 01/06/2014 4:20:12 PM  
  
Name      : nvcompiler.dll  
Length    : 25256224  
LastWriteTime : 11/15/2013 9:52:30 AM  
  
Name      : nvoglv64.dll  
Length    : 26940704  
LastWriteTime : 11/15/2013 9:52:56 AM
```

When you leverage the pipe command, you have the ability to find specific data pertaining to an object. In this example, you search all of

the files in `c:\windows\system32` to determine whether there are any files that have a size greater than 20 MB. You were able to pipe the `get-childitem` cmdlet to the `where` operator with the selection criteria `$_.length` is greater than 20MB. These results were placed in the `$largeFiles` variable. From there, you use the `count()` method to count the number of files that are larger than 20MB. You then print to screen the text There are `$count` files over 20MB. You also print to the screen Files over 20MB in `c:\Windows\System32\` : to provide text for the following piped command. You then need to take the results in the `$largeFiles` variable and pipe the results to the `select-object` command to select the `name`, `length`, and `lastwritetime`. Finally, you pipe those results to the `format-list` command, which provides a formatted list of results.

Summary

This chapter explores the many methods you can use to manipulate and parse data with PowerShell. You learned techniques that will help you better work with data and provide a richer experience for individuals using your scripts.

The string manipulation section taught you many different methods to work with strings. This included changing the case, splitting and replacing strings, counting and trimming strings, searching strings, and viewing substrings. The number manipulation section taught you how to use mathematical operations within PowerShell. This section displayed different ways to format numbers, round numbers, and calculating complex mathematical operations. The date time section of this chapter provided you with tools to use when you need to gather date and time information from a system. You learned how to format the `get-date` cmdlet, manipulate the results, and add or subtract from date values.

This chapter also explored the forcing of data types while working with variables. This section provided examples of different data types that are available to use, and provided an example on how to force a specific data type. You then wrapped up with an explanation on piping and how to construct proper piping clauses. It explored how to leverage piping to sort data after a result is returned and provided examples of piping with and without a data sort.

Data parsing and manipulation is essential to successful scripting with PowerShell. Without using these manipulation techniques in this chapter, you may be overly complicating your scripts. While it may take time to fully learn these techniques, it's essential to become a good PowerShell scripter. In the next chapter, you'll learn how the manipulated data you generate can be correlated to see whether it matches certain criteria. You'll learn that the comparisons are done with the use of comparison operators.

Chapter 3. Comparison Operators

PowerShell comparison operators are used to validate data present within your scripts. These operators enable you to compare data and execute code based on the data. This makes PowerShell an extremely effective tool to use for processing complicated data with the available comparison operators.

In this chapter, you will learn the following concepts:

- Comparison operator basics
- Equal and not equal comparison operators
- Greater than and less than comparison operators
- Contains, like, and match comparison operators
- -AND / -OR comparison operators
- Best practices for comparison operators

Comparison operator basics

When you are using comparison operators, you are creating expressions that evaluate to either `True` or `False`. In programming, this is known as Boolean. In the simplest form, you are asking PowerShell to evaluate similarities or dissimilarities between two items. Based on the findings from that expression, it will return `True` or `False`. When the whole expression returns `False`, PowerShell doesn't continue to process items in the statement. When the whole expression returns `True`, PowerShell will proceed forward into the statement and execute the code within the statement.

Of the many built-in variables that PowerShell has, there are two built-in Boolean variables. These two variables are `$True` and `$False`. When you call `$True`, it implies that the value is Boolean and is set to `True`. When you call `$False`, it implies that the value is Boolean and is set to `False`.

A script that shows how to use basic comparison operators would look like this:

```
$TrueVariable = $True
$FalseVariable = $False
if ($TrueVariable) { Write-Host "Statement is True." }
if ($FalseVariable) { Write-Host "Statement is False." }
```

The output of this command is shown in the following screenshot:

```
PS C:\> $TrueVariable = $True
PS C:\> $FalseVariable = $False
PS C:\> if ($TrueVariable) { Write-Host "Statement is True." }
Statement is True.
PS C:\> if ($FalseVariable) { Write-Host "Statement is False." }
PS C:\>
```

This example displays how to do a basic Boolean comparison. You first start by declaring `$TrueVariable` and setting it equal to `True`. You then declare `$FalseVariable` and set it to `False`. You then create an `if` statement to evaluate the expression `$TrueVariable` to see whether it evaluates to `True`. Since the variable is set to `True`, it will evaluate to `True` and continue to process the remaining items in the statement. PowerShell will print to the screen the message `Statement is True`. You then create another `if` statement to evaluate the expression `$FalseVariable` to see whether it evaluates to `True`. As the variable is set to `False`, the expression will evaluate to `False` and stop processing the statement.

Equal and not equal comparison

The most basic and most used comparison operator is *equal to* (-eq). This operator is flexible in nature as it can be used for strings, integers, and objects. The -eq operator is used by calling `$value1` *is equal to* `$value2`. When the -eq operator evaluates the statement, it will return a Boolean value of either `True` or `False`. If the expression evaluates to be `True`, PowerShell will continue to proceed to execute the code.

A script that shows how to use equal comparison operators would look like this:

```
$value1 = "PowerShell"  
$value2 = "PowerShell"  
if ($value1 -eq $value2) { Write-Host "It's Equal!" }
```

The output of this is shown in the following screenshot:

```
PS C:\> $value1 = "PowerShell"  
PS C:\> $value2 = "PowerShell"  
PS C:\> if ($value1 -eq $value2) { Write-Host "It's Equal!" }  
It's Equal!
```

From the preceding example, you will see that the equal comparison operator determines that `$value1` is equal to `$value2` and it writes to the screen `It's Equal!`. In the instance that you want to determine whether two values are not equal, you can use the -ne operator. This does the inverse of the -eq operator.

A script that shows how to use not equal comparison operators would look like this:

```
$value1 = "PowerShell"  
$value2 = "POSH"  
if ($value1 -ne $value2) { Write-Host "Values Are Not Equal" }
```

The output of this command is shown in the following screenshot:

```
PS C:\> $value1 = "PowerShell"
PS C:\> $value2 = "POSH"
PS C:\> if ($value1 -ne $value2) { Write-Host "Values Are Not Equal" }
Values Are Not Equal
```

When you run the preceding script, PowerShell will determine that \$value1 and \$value2 are not equal. The script will write to the screen the message Values Are Not Equal. While scripting, it is important to minimize the use of the *not equal* -ne operator. When you start layering *is equal to* and *not equal to* in your scripts, the logical complexity of the script significantly increases. This is why it is recommended that beginners should typically only use the -ne operator in instances where a value cannot equal a specific value and every other value is acceptable.

A script that shows how to use "not equal" comparison operators would look like this:

```
$value = "This is a value."
$length = $value.length
If ($length -ne 0) { Write-Host "The variable has data in it.
Do this action" }
```

The output of this command is shown in the following screenshot:

```
PS C:\> $value = "This is a value."
PS C:\> $length = $value.length
PS C:\> If ($length -ne 0) { Write-Host "The variable has data in it. Do this action" }
The variable has data in it. Do this action
```

The preceding example displays the proper use of the -ne operator for best practices. This script counts the characters in \$value, and if the length of the variable is not equal to 0, the script will write to the console The variable has data in it. Do this action. In your scripting, you will want to follow suit where you use -ne for verification that the data is *not valid* before continuing with your script.

Greater than and less than comparison

PowerShell has two operators to compare two values to determine whether they are greater than (-gt) or less than (-lt) each other. This is not just limited to numbers, but also has the ability to compare dates and times as well. These are helpful in instances where you need to compare file sizes or modification dates on files.

A script that shows how to use the "less than" comparison operator would look like this:

```
$number1 = 10
$number2 = 20
If ($number1 -lt $number2) { Write-Host "Value $number1 is less than $number2" }
```

The output of this command is shown in the following screenshot:

```
PS C:\> $number1 = 10
PS C:\> $number2 = 20
PS C:\> If ($number1 -lt $number2) { Write-Host "Value $number1 is less than $number2" }
Value 10 is less than 20
```

In the preceding example, you set the \$number1 variable to 10 and the \$number2 variable to 20. You then use the "less than" (-lt) operator to determine whether the \$number1 variable is less than \$number2. Since this is a true statement, the console outputs the message Value \$number1 is less than \$number2.

A script that shows how to use the "greater than" comparison operator would look like this:

```
$olddate = Get-Date
Start-Sleep -seconds 2
$newdate= Get-Date
If ($newdate -gt $olddate) { Write-Host "Value $newdate is greater than $olddate" }
```

The output of this command is shown in the following screenshot:

```
PS C:\> $olddate = Get-Date
PS C:\> Start-Sleep -seconds 2
PS C:\> $newdate= Get-Date
PS C:\> If ($newdate -gt $olddate) { Write-Host "Value $newdate is greater than $olddate" }
Value 02/17/2015 12:51:46 is greater than 02/17/2015 12:51:44
```

In this script, you start by setting the `$olddate` variable to the current date and time. The `start-sleep` cmdlet is then used to pause the script for 2 seconds. When the script continues, you set `$newdate` and time variable to 2 seconds later. By using the greater than (`-gt`) operator, you determine that the values are different and that the `$newdate` value is greater than the `$olddate` value.

Tip

In addition to "greater than" and "less than" operators, you also have the option to compare "greater" or "equal" (`-ge`) and "less" or "equal" (`-le`). These comparison operators can be handy when creating counters or loops that require you to increment a number until it equals or is greater than a specific value.

Contains, like, and match operators

The `-contains`, `-like`, and `-match` operators are very similar in function. While they all compare data, they all have their own purpose in your scripts. Each of these operators are case-insensitive. This means that when you are searching for items using these operators, they will match all instances of the value in the expression. In instances where you need the search to be case-specific, you can append `c` in front of the operator to force case sensitivity. These would look like `-ccontains`, `-clike`, and `-cmatch`. To force case insensitivity, you can also append `i` in front of the operator. These would look like `-icontains`, `-ilike`, and `-imatch`.

Each of these operators also has an inverse operator that is formed by appending the word "not" in front of the operator. Examples of these operators include `-notcontains`, `-notlike`, and `-notmatch`. You may also append case sensitivity and case insensitivity to these operators.

The `-contains` operator looks for an exact match to a value in an expression. It will then return a `True` and `False` result. The `-contains` operator is flexible as it can evaluate the individual values in an array with a single expression. This allows you to create efficiency in your code by evaluating more than one item per line of code.

A script that shows how to use the `-Contains` comparison operator would look like this:

```
$myarray = "this", "is", "my", "array"
If ($myarray -contains "this") { Write-Host "The array contains
the word: this" }
If ($myarray -notcontains "that") { Write-Host "The array does
not contain the word: that" }
```

The output of this command is shown in the following screenshot:

```
PS C:\> $myarray = "this", "is", "my", "array"
PS C:\> If ($myarray -contains "this") { Write-Host "The array contains the word: this" }
The array contains the word: this
PS C:\> If ($myarray -notcontains "that") { Write-Host "The array does not contain the word: that" }
The array does not contain the word: that
```

In the preceding example, you create an array with four values in it. You then use the `-contains` operator to determine whether the array has the `this` value. As the array does have this value, it then evaluates the statement to be `True`, and proceeds to write to the console the message `The array contains the word: this`. The second part of this evaluation is to check to see whether `$myarray` does not contain the word `that` by using the `-notcontains` operator. Since `$myarray` does not have the word `that`, it proceeds to write to the console `The array does not contain the word: that`.

The `-like` comparison operator is different than the `-contains` operator. The `-like` operator requires that both sides of the expression should be evaluated to match the full string. You can quickly determine whether there are values that are close to a value you are looking for, which will then return `True` or `False`. This is why the `-like` comparison operator typically uses wildcard characters designated by an asterisk (*) or question mark (?). It provides the flexibility to quickly search a variety of values using a single expression. The asterisk wildcard designates that the expression can match any values before or after the stated word, depending on where the asterisk is placed. The question mark allows you to match any values present between two strings. For example, you can use `-like "myfile?.txt"`, which will match any value that starts with `myfile` and ends with the extension `.txt`. Any values between those characters will be returned as `True`.

A script that shows how to use the `-like` comparison operator would look like this:

```
$myexample = "This is a PowerShell example."
If ($myexample -like "*shell*") { Write-Host "The variable has
a word that is like shell" }
If ($myexample -notlike "*that*") { Write-Host "The variable
```

```
doesn't have a word that is like that" }
```

The output of this command is shown in the following screenshot:

```
PS C:\> $myexample = "This is a PowerShell example."
PS C:\> If ($myexample -like "*shell*") { Write-Host "The variable has a word that is like shell" }
The variable has a word that is like shell
PS C:\> If ($myexample -notlike "*that*") { Write-Host "The variable doesn't have a word that is like that" }
The variable doesn't have a word that is like that
```

The preceding script displays the string `$myexample`, for which you search for the value of `shell`. As the value is part of another word, you need to append the wildcard character on both sides of the word "shell". When you search for the word `*shell*`, including the wildcard characters, the result returns true. The console then outputs the message The variable has a word that is like shell. When you execute the second comparison using the `-notlike` operator, you are able to search the string for words that are not like `that`. Since you use the wildcards on each side of the word, it does a secondary comparison to make sure that there aren't partial values in the variable that reflect `that`. Since there are no values in the variable that evaluate to be like `that`, it outputs to the screen the message The variable doesn't have a word that is like shell.

Tip

You have to use an asterisk (*) on both sides of the search evaluation criteria as PowerShell interprets every character in the sentence as a value. While inherently you may break apart each word in the variable as separate values, PowerShell sees it as one contiguous group of characters. To find a substring of a variable and have it evaluate to "True" using the `-like` operator, you will need to use an asterisk (*) on both sides.

The match comparison operator uses regular expressions to match information between two variables. The `-match` operator is unique in the fact that it autopopulates a variable named `$matches` with the word that matches your search. This is helpful in the instance where you need to only retrieve objects that match a certain criteria. With "match", you

can also leverage the use of regular expressions to match criteria to a variable.

A script that shows how to use the `-match` and `-notmatch` comparison operators would look like this:

```
$myexample = "The network went down."
If ($myexample -match "[o]") { Write-Host "The variable matched
the letter o. (Contains two o's)" }
$matches
If ($myexample -notmatch "[U]") { Write-Host "The variable does
not match U. (Doesn't have a U)" }
```

The output of this command is shown in the following screenshot:

```
PS C:\> $myexample = "The network went down."
PS C:\> If ($myexample -match "[o]") { Write-Host "The variable matched the letter o. (Contains two o's)" }
The variable matched the letter o. (Contains two o's)
PS C:\> $matches
Name          Value
----          -----
o

PS C:\> If ($myexample -notmatch "[U]") { Write-Host "The variable does not match U. (Doesn't have a U)" }
The variable does not match U. (Doesn't have a U)
```

The preceding example creates a new variable named `$myexample` with the string value of `The network went down..` You then compare the `$myexample` variable to the regular expression `[o]` or one that contains an instance of `o` to see that it's a match. Since `o` exists at least once contained in `$myexample`, the expression returns `True` and the console outputs the message `The variable matched the letter o. (Contains two o's)`. After you make that comparison, you then display the contents of the `$matches` variable. You will see that the `$match` variable autopopulates with the value of an index of `0` and name of `o`. The last part of the script is an evaluation to see whether `$myexample` does not match the regular expression of `[U]` or does not contain an instance of `U`. Since the variable does not contain an instance of the letter `U`, it evaluates to be `True` and writes to the console the message `The variable does not match U. (Doesn't have a U)`.

And / OR comparison operators

The `-and` and `-or` comparison operators are used to evaluate multiple expressions present within a single line of code. These are used to see whether two or more expressions evaluate to be `True`. The `-and` comparison operator mandates that both evaluations must evaluate to be `True` to proceed in the statement. This means that `expression1` and `expression2` must be `True` to continue. The `-or` comparison operator only requires one of the two expressions to be `True`. This means that `expression1` or `expression2` can be `True` to continue. As you are learning PowerShell, you will want to use caution while using the `-and` and `-or` comparison operators as they can quickly complicate the logic of your scripts.

A script that shows how to use `-and` and `-or` comparison operators would look like this:

```
$myvar = $True
$myothervar = $False
If ($myvar -eq $True -AND $myothervar -eq $False) { Write-Host
"Both statements evaluate to be True" }
If ($myvar -eq $True -OR $myothervar -eq $True) { Write-Host
"At least one statement evaluates to be True" }
```

The output of this is shown in the following screenshot:

```
PS C:\> $myvar = $True
PS C:\> $myothervar = $False
PS C:\> If ($myvar -eq $True -AND $myothervar -eq $False) { Write-Host "Both statements evaluate to be True" }
Both statements evaluate to be True
PS C:\> If ($myvar -eq $True -OR $myothervar -eq $True) { Write-Host "At least one statement evaluates to be True" }
At least one statement evaluates to be True
```

The preceding example briefly displays how to use the `-and` operator and the `-or` operator. In this example, you create two different variables. You then check to see whether `$myvar` equals `True`, which evaluates to be `True`. You evaluate whether `$myothervar` is equal to `False`, which evaluates to be `True`. In order for the `-and` operator to be successful, both statements have to evaluate to be `True` in the evaluation criteria.

Since both the statements evaluate to be True, the console outputs the message Both statements evaluate to be True. Even though the \$myothervar variable is set to False, the evaluation to see whether that variable is set to False makes that statement True.

The second statement you evaluate is when either \$myvar or \$myothervar equals True by using the -OR operator. Like the first evaluation, the first variable evaluates to be True. However, the second variable evaluates to be False. Since the -or operator only requires one of the two statements to be True, the entire statement evaluates to be True. The console will output the message At least one statement evaluates to be True.

Best practices for comparison operators

PowerShell offers many different comparison operators for use within your scripts. It is easy to start building overly complex scripts by overusing comparison operators or by evaluating items that you may not have to use in PowerShell functioning. Refer to the following guidelines to stick to when you are developing your scripts. These will help you avoid overuse of comparison operators:

- **Assume the script is designed to proceed:** When you assume your script is designed to proceed to the next step, you can reduce the number of comparison operators you use. If you expect a value to be `True`, only make a statement to catch whether the statement is `False`. Don't check to see whether the statement is `True`, as PowerShell is designed to sequentially proceed anyway to the next step.
- **Avoid double negative statements:** When you are developing your code, avoid the use of double negative statements. Avoid checking to see whether a value *does not equal* `False`. What you're really trying to do is check to see whether a statement evaluates to be `True`. Double negatives can be confusing to you and other developers reading your code.
- **Stay positive (`True`) while you're coding:** Always attempt to avoid the use of *not* and negative evaluation statements. While there can be a place for the *not* based operators, try to create code that evaluates when statements are `True`. The *not* based operators grow significantly in complexity when used with regular expressions and can be confusing to you and other developers reading your code.

Summary

This chapter explored the many methods with which you can use PowerShell operators. You started by learning the comparison operator basics. You then learned about the *equal* and *not equal* and *greater than* and *less than* comparison operators. You learned that you can use these operators to compare numbers, strings, dates, and times. You then proceeded to explore the `-contains`, `-like`, and `-match` operators. You learned that you can add *not* to these operators to create the inverse of the operator. You also understood that you can add *c* for case sensitivity and *i* for case insensitivity to the comparison operators. You also saw how to join multiple operators using the `-and` / `-or` operators.

The chapter ends by providing the best practices for the implementation of comparison operators. By the end of this chapter, you should be proficient in using comparison operators, know what to avoid, and be well on your way to evaluating variables and arrays. In the next chapter, you will explore how you can create code that can be called multiple times and leverage comparison operators with functions, loops, switches, and methods.

Chapter 4. Functions, Switches, and Loops Structures

When you are scripting in PowerShell, you will find that a lot of your coding efforts will require the code to be repeated multiple times in the same script. While repeating the same code may help you accomplish the task, there are many other options for coding more efficient scripts. This chapter explores different techniques for which you can reuse code instead of repeating the same code segments within the same script.

In this chapter, you will learn about the following concepts:

- Creation of functions
- Creation of loops
- Creation of switches
- Combining the use of functions, switches, and loops
- Best practices for functions, switches, and loops

Functions

When you need to query or execute code more than once, the general rule is that you should create a function to perform the action. Functions are blocks of reusable code, which you can execute multiple times by calling the function's name. You must place a function near the beginning or top of the script. This allows PowerShell to interpret the whole function before you use it later in the code. All other code, including invoking the functions, should follow the functions section. If you call a function that has not yet been parsed by PowerShell, it will throw an exception stating that no such cmdlet or function exists.

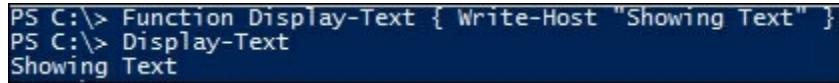
Function names can be any word or set of words; however, it is recommended to name the function similar to the verb-noun cmdlet naming syntax. To create a function, you need to use the word `Function` and declare a function name like `display-text`. You then need to enclose the repeatable commands in curly brackets after the function

name.

The proper syntax of a function looks like this:

```
Function Display-Text { Write-Host "Showing Text" }
Display-Text
```

The output of this command is shown in the following screenshot:



```
PS C:\> Function Display-Text { Write-Host "Showing Text" }
PS C:\> Display-Text
Showing Text
```

This example displays how to properly declare a function. You first call the `Function` command with the `Display-Text` function name. You then place `Write-Host "Show Text"` in the curly brackets after declaring the function name. You then call the function by typing `Display-Text`. After executing the script, the console will print to the screen the message `Showing Text`.

Functions also allow you to *pass in data* for processing. One of the methods to *pass in data* into a function is to declare variables after the function name in parentheses. This function will then be able to use those variables and the data in those variables within itself. If you want to pass in multiple arguments into a function, you can separate each variable with a comma.

The format to declare a function with parameters in parentheses looks like this:

```
Function Display-Text($variable1,$variable2) {
    Write-Host "First Function Argument: $variable1"
    Write-Host "Second Function Argument: $variable2"
}
Display-Text "Hello" "Readers"
```

The output of this is shown in the following screenshot:

```
PS C:\> Function Display-Text($variable1,$variable2) {  
>> Write-Host "First Function Argument: $variable1"  
>> Write-Host "Second Function Argument: $variable2"  
>> }  
>> Display-Text "Hello" "Readers"  
>>  
First Function Argument: Hello  
Second Function Argument: Readers
```

This example displays how to properly declare a function with the parameter in parentheses. You first call the `Function` command with the `Display-Text` function name. You then place the variables, `$variable1` with a comma and `$variable2` in parentheses, before the curly brackets. Inside the curly brackets, you declare `Write-Host "First Function Argument: $variable1"` and `Write-Host "Second Function Argument: $variable2"`. You then call the function by typing `Display-Text` with the arguments of `Hello` and `Readers`. After executing the script, the console will print to the screen `First Function Argument: Hello` and `Second Function Argument: Readers`.

Another method to *pass in data* to a function is through the use of a parameter block of `Param`. `Param` takes in whatever data you pass into the function and stores that data in declared variables. If you want to pass in multiple arguments into a function, you can separate each variable with a comma. When you are declaring parameters using this method, `Param` needs to be the first item declared after the open curly bracket in a function.

The format to declare a function with `param` looks like this:

```
Function Display-Text { Param($variable1, $variable2)  
Write-Host "First Function Argument: $variable1"  
Write-Host "Second Function Argument: $variable2"  
}  
Display-Text "Hello" "Readers"
```

The output of this command is shown in the following screenshot:

```
PS C:\> Function Display-Text { Param($variable1, $variable2)
>> Write-Host "First Function Argument: $variable1"
>> Write-Host "Second Function Argument: $variable2"
>>
>> Display-Text "Hello" "Readers"
>>
First Function Argument: Hello
Second Function Argument: Readers
```

This example displays how to properly declare a function with parameters in a `Param` block. You first call the `Function` command with the `Display-Text` function name. You then call the `Param` block as the first command inside the curly brackets. Inside the `Param` block, you declare the variables `$variable1` with a comma and `$variable2`. After the `Param` block, you declare `Write-Host "First Function Argument: $variable1"` and `Write-Host "Second Function Argument: $variable2"`. You then call the function by typing `Display-Text` with the arguments `Hello` and `Readers`. After executing the script, the console will print to the screen `First Function Argument: Hello` and `Second Function Argument: Readers`.

The `Param` block is special as it can also accept additional decorators when declaring the variables. The `[Parameter()]` decorator allows you to include additional arguments that enable you to validate variables and even provide help information for variables in functions. When you declare the `Mandatory` argument and set it equal to `$True`, it will require that the variable is used in the function to continue. If you set the `Mandatory` argument to `$False`, it will not be required when using the function. You can also call the `Position` argument, which declares what position the variable will be declared. This means that if you set the `Position` argument to `1`, it must be the first argument passed into the function. If you don't use the `Position` argument, you will only be able to pass in the variables using parameter that references the variable name. Another popular argument is the `HelpMessage` argument, which enables you to declare a help message for the individual arguments being passed in. This message is what is displayed in the console when mandatory arguments are missing when a function is being executed. To add multiple parameter arguments in a decorator, you can separate the arguments with commas.

The format to declare a function with `Param` looks with the `[Parameter()]` decorator and parameter arguments looks like this:

```
Function Display-Text {
    #Declare the Parameter Block
    Param(
        #Set The First Parameter as Mandatory with a Help Message
        [Parameter(Mandatory=$True,HelpMessage="Error: Please Enter A Computer Name")]$computername,
        #Set the Second Parameter as Not Mandatory
        [Parameter(Mandatory=$False)]$Message
    )
    Write-Host "First Mandatory Function Argument: $computername"
    Write-Host "Second Function Argument: $Message"
}
Display-Text -computername "MyComputerName" "MyMessage"
Display-Text
```

The output of this command is shown in the following screenshot:

```
PS C:\> Function Display-Text {
>>> #Declare the Parameter Block
>>> Param(
>>>     #Set The First Parameter as Mandatory with a Help Message
>>>     [Parameter(Mandatory=$True,HelpMessage="Error: Please Enter A Computer Name")]$computername,
>>>     #Set the Second Parameter as Not Mandatory
>>>     [Parameter(Mandatory=$False)]$Message
>>>
>>>     Write-Host "First Mandatory Function Argument: $computername"
>>>     Write-Host "Second Function Argument: $Message"
>>>
>>>     Display-Text -computername "MyComputerName" "MyMessage"
>>>     Display-Text
>>
First Mandatory Function Argument: MyComputerName
Second Function Argument: MyMessage

cmdlet Display-Text at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
computername: !?
Error: Please Enter A Computer Name
computername: _
```

This example displays how to create a function using `param` with the `[Parameter()]` decorator and parameter arguments. You first call the `Function` command with the `Display-Text` function name. You then call the `Param` block as the first command inside the curly brackets. Inside the `Param` block, you declare the several parameter arguments for the variables. The first argument you call is the `[Parameter]` decorator, and

the `Mandatory=$True` attribute. You then include a comma to accept the second attribute of `HelpMessage="Error: Please Enter A Computer Name"`. You then close the parameter decorator and complete the first `Param` item by defining the `$computernname` variable.

You include a comma to create a second `Param` item. This `Param` item uses the `[Parameter]` decorator and the `Mandatory=$False` attribute. You close the parameter decorator and complete the second `Param` item by defining the `$message` variable. You then close the `Param` block.

After the `Param` block, you declare `Write-Host "First Mandatory Function Argument: $computernname"` and `Write-Host "Second Function Argument: $Message"`. You then call the function by typing `Display-Text` with the arguments `-computername "MyComputerName"` and `"MyMessage"`. You also call `Display-Text` without any arguments.

After executing the script, the console will first print to the screen First Mandatory Function Argument: MyComputerName and Second Function Argument: MyMessage. When the script executes the second `Display-Text`, however, it will print on the screen cmdlet `Display-Text` at command pipeline position 1 Supply values for the following parameters: Type `!?` for help.. It will then prompt for the `computernname` argument. If you type `!?` and press `Enter`, you will see the `HelpMessage` attribute displayed in the console with the message `Error: Please Enter A Computer Name`. It will then prompt for the `computernname` argument again until you enter a value.

Tip

Additional detailed information on advanced parameters for functions can be found on TechNet at <https://technet.microsoft.com/en-us/library/dd347600.aspx>.

Functions allow you to *pass back data* to the section of the script that called the function in the first place. One of the methods with which you can achieve this is with the use of the `return` command. For example, if after execution of a function you want to pass back the value of `$True`, you can state `return $True`. The section of the script that executed the

command will then be able to use the value of \$True to execute on. You may also use the `write-output` cmdlet, which acts like the return command and *passes back* the values to the script. You could also choose the piping method to *pass back data*. To use the piping method, you take the output from the function and pipe it to a cmdlet or another section of code.

The format to declare functions that return values to the script looks like this:

```
Function Create-WarningMessage {  
    $MyError = "This is my Warning Message!"  
    $MyError  
}  
Function Create-Message { Return "My Return message." }  
Function Create-Message2 { Write-Output "My Write-Output  
message." }  
Create-WarningMessage | Write-Warning  
Create-Message  
Create-Message2
```

The output of this command is shown in the following screenshot:

```
PS C:\> Function Create-WarningMessage {  
    >> $MyError = "This is my Warning Message!"  
    >> $MyError  
    >>  
    >> }  
    >> Function Create-Message { Return "My Return message." }  
    >> Function Create-Message2 { Write-Output "My Write-Output message." }  
    >> Create-WarningMessage | Write-Warning  
    >> Create-Message  
    >> Create-Message2  
    >>  
    >> WARNING: This is my Warning Message!  
    >> My Return message.  
    >> My Write-Output message.
```

This example displays how to declare functions that return values to the script. You first call the `Function` command with the `Create-WarningMessage` function name. Inside the curly brackets, you create a variable named `$MyError` and set it equal to `This is my Warning Message`. You then call the `$MyError` variable and close the function. You create a second function by using the `Function` command with the `Create-Message` function name. Inside the curly brackets, you use the `Return` command with the message `My Return message`. Finally, you

create a third function by using the `Function` command with the `Create-Message2` function name. Inside the curly brackets, you use the `Write-Output` cmdlet with the text `My Write-Output message`.

When you run the script, you first call the `Create-WarningMessage` function and pipe it to `Write-Warning`. When you do this, the output from `Create-WarningMessage` of `This is my Warning Message!` is passed to the `Write-Warning` via the pipeline, and a warning message of `WARNING: This is my Warning Message!` is printed to the console. You then call the `Create-Message` function, which returns from the function and prints to the screen `My Return message`. Finally, you call the `Create-Message2` function, which passes back the `Write-Output` cmdlet message and prints to the screen `My Write-Output message`.

Tip

If you need to exit a function, you can simply use the `return` command, which gracefully exits the function. This avoids having to stop the whole script by using the `EXIT` or `BREAK` commands.

Looping structures

PowerShell provides a variety of looping structures for evaluating and executing objects. Loops are helpful in situations where you need to take in an array of objects and process the individual values in the array. Subsequently, loops are also helpful in situations where you need to wait for a specific value within an array before proceeding in the script.

There are four main looping structures in PowerShell. These looping structures include `Do/While`, `Do/Until`, `ForEach`, and `For`. The `Do/While` looping structure is used to execute a task when a value doesn't equal a specific value. The inverse of this looping structure is `Do/Until`, where it will keep looping the structure until a value equals a specific value. `ForEach` is a looping structure that allows you to process each individual object in an array or set of objects. The `For` loop is typically used to execute a task a set number of times.

To create a new `Do/While` looping structure, you first start by declaring the `Do` command. You then place the PowerShell commands that you want to repeat in curly brackets. After closing the curly brackets, you declare the `While` command with a conditional statement. The condition typically leverages a comparison operator and you tell the loop to repeat when the statement equals a specific condition. Once the evaluation of the condition no longer returns `True`, the loop will stop.

The format of a `Do/While` looping structure looks like this:

```
$x = 1
$myVar = $False
Do {
    If ($x -ne "4") {
        Write-Host "This Task Has Looped $x Times"
    }
    If ($x -eq "4") {
        $myVar = $True
        Write-Host "Successfully executed the script $x times"
    }
    $x++
}
```

```
While ($myVar -eq $False)
```

The output of this is shown in the following screenshot:

```
PS C:\> $x = 1
PS C:\> $myVar = $False
PS C:\> Do {
>>     If ($x -ne "4") {
>>         Write-Host "This Task Has Looped $x Times"
>>     }
>>     If ($x -eq "4") {
>>         $myVar = $True
>>         Write-Host "Successfully executed the script $x times"
>>     }
>>     $x++
>> }
>> While ($myVar -eq $False)
>>
This Task Has Looped 1 Times
This Task Has Looped 2 Times
This Task Has Looped 3 Times
Successfully executed the script 4 times
```

The preceding script displays the proper usage of the `Do/While` loop structure. The script starts by declaring a variable `$x` equal to 1. The `$x` variable designates that it is the first time you are executing the script. You will then declare the `$myVar` equal to `False` to allow the script to execute while the variable is `False`. The `Do` clause will then execute while `$myVar` equals `False`. With each iteration of the loop, the script will evaluate whether the `$x` variable equals 4. If it doesn't equal 4, it will write to the console `This task has Looped $x Times`. It will increment `$x` by one value designated by the `$x++` command and restart from the beginning of the loop. When `$x` equals 4, the script will set `$myVar` value to `True` and write to the console the message `Successfully executed the script $x times`. The loop will evaluate `$myVar` and determine that it no longer equals `False` and exit the loop.

To create a new `Do/Until` looping structure, you first start by declaring the `Do` command. You then place the PowerShell commands that you want to repeat in curly brackets. After closing the curly brackets, you declare the `Until` command with a conditional statement. The condition typically leverages a comparison operator and you tell the loop to repeat until the statement equals a specific condition. Once the evaluation of the condition no longer returns `False`, the loop will stop.

Tip

When you are creating looping structures, it's inevitable that you will accidentally create an infinite looping structure. When you do, you may be flooding your console with text or create a large amount of data. To pause a loop, press *Pause* on your keyboard. If you want to continue, you can hit *Enter* on the keyboard. To completely exit a loop, you can press the key combination of *Ctrl + C* in the console window. This will break the looping structure.

The format of a `Do/Until` looping structure looks like this:

```
$x = 1
$myVar = $False
Do {
    If ($x -ne "4") {
        Write-Host "This Task Has Looped $x Times"
    }
    If ($x -eq "4") {
        $myVar = $True
        Write-Host "Successfully executed the script $x times"
    }
    $x++
}
Until ($myVar -eq $True)
```

The output of this is shown in the following screenshot:

```
PS C:\> $x = 1
PS C:\> $myVar = $False
PS C:\> Do {
>>     If ($x -ne "4") {
>>         Write-Host "This Task Has Looped $x Times"
>>
>>     If ($x -eq "4") {
>>         $myVar = $True
>>         Write-Host "Successfully executed the script $x times"
>>
>>     }
>>     $x++
>> }
>> Until ($myVar -eq $True)
>>
This Task Has Looped 1 Times
This Task Has Looped 2 Times
This Task Has Looped 3 Times
Successfully executed the script 4 times
```

The preceding script displays the proper usage of the `Do/Until` loop structure. The script starts by declaring a variable `$x` equal to `1`. The `$x` variable designates that it is the first time you are executing the script. You will then declare `$myVar` equal to `False` to allow the script to execute while the variable is `False`. The `Do` clause will then execute until `$myVar` equals `True`. With each iteration of the loop, the script will evaluate whether the `$x` variable equals `4`. If it doesn't equal `4`, it will write to the console `This task as Looped $x Times.` It will increment `$x` by one value designated by the `$x++` command and restart from the beginning of the loop. When `$x` equals `4`, the script will set the `$myVar` value to `True` and write to the console `Successfully executed the script $x times.` The loop will evaluate `$myVar` and determine that it no longer equals `True` and exit the loop. You will see that the `Do/Until` loop structure is declared exactly as the previous script; however, PowerShell interprets the `Until` statement as a conditional statement to continue *until something equals a value*.

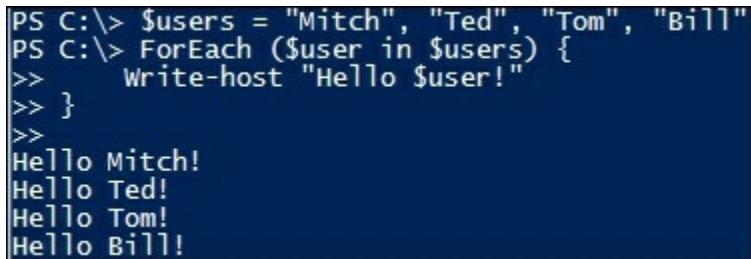
The `ForEach` loop structure has a very simple construct. The `ForEach` looping structure is declared by calling `ForEach`. You then specify parentheses containing a variable, the word `in`, and typically a second variable that contains an array of data. This may look like `($object in $array)`. While the `$array` variable typically contains a set of objects, the `$object` variable is considered the *processing variable*. This variable enables you to access each object in the array and its properties. After you declare the variables in the parentheses, you place the PowerShell code you want to repeat in curly brackets. In the instance that you want to interact with the individual objects in the array, you can leverage the processing variable in your PowerShell code.

While creating the `ForEach` loop for processing variables and arrays, it's important to name the variables reflective of what you are processing. If you had a list of account numbers, you could create variables that reflected `ForEach ($account in $accountNumbers)`. This will reduce confusion while reading your scripts.

The format of a `ForEach` looping structure looks like this:

```
$users = "Mitch", "Ted", "Tom", "Bill"
ForEach ($user in $users) {
    Write-host "Hello $user!"
}
```

The output of this is shown in the following screenshot:



```
PS C:\> $users = "Mitch", "Ted", "Tom", "Bill"
PS C:\> ForEach ($user in $users) {
>>     Write-host "Hello $user!"
>> }
>>
Hello Mitch!
Hello Ted!
Hello Tom!
Hello Bill!
```

In this example, you define an array named `$users` with the individual values of `Mitch`, `Ted`, `Tom`, and `Bill`. You then declare the `ForEach` loop with a processing variable of `$user`. This loop will then process each `$user` in the array `$users` and write to the console the message `Hello $user!`. The `$user` variable will be reflecting the current value of the current object that the loop is processing.

The `For` looping structure has a slightly more complex construct. You first start by declaring the `For` command. You then declare three required sections of code separated by semicolons and enclose these sections in parentheses. The first section of code is declaring a variable that will interact with the looping structure. This typically is a number that is incremented or decreased as the looping structures proceeds through the loops. This variable must contain a value, otherwise the looping structure will not proceed. This value can be either defined before you enter the looping structure, or when you are declaring the looping structure itself.

The second section of code is the conditional statement, which tells the loop to continue while this statement is `True`. Once the statement is `False`, the loop terminates. The last section of code tells the loop to either increment or decrease the first variable and by how many. You can use shorthand to increase a variable by entering `$variable++` to

increase the value `1`, or you can decrease the value by performing a math operation like `$variable - 1`. After enclosing the required sections of the `For` loop structure, you enclose the code you want to repeat in curly brackets.

The format of a `For` looping structure may look like this:

```
For ($x = 1; $x -lt "5"; $x++) {  
    write-host "Hello World! Loop Attempt Number: $x"  
}
```

The output is shown in the following screenshot:

```
PS C:\> For ($x = 1; $x -lt "5"; $x++) {  
    >> write-host "Hello World! Loop Attempt Number: $x"  
    >>  
    >>  
    Hello World! Loop Attempt Number: 1  
    Hello World! Loop Attempt Number: 2  
    Hello World! Loop Attempt Number: 3  
    Hello World! Loop Attempt Number: 4
```

This example displays how to properly use a `For` looping structure. You first start by declaring the `For` command. You then declare the required sections for the structure; you start by defining `$x` equal to the value `1`, which starts the first loop at the value of `1`. You then declare the conditional statement of loop while `$x` is less than `5`. In the last required section, you declare `$x++`, which increments the `$x` variable by `1` in every loop throughout the structure. You then declare the PowerShell command `Write-host "Hello World! Loop Attempt Number: $x"` in curly brackets. When you run this script, the `For` looping structure will loop `4` times writing to the console the message `Hello World! Loop Attempt Number: $x`, where `$x` equals the iteration of the script's loop.

Tip

Also, it is important to remember that the `Do/While`, `Do/Until`, and `For` loop structures do not increment the `$x` variable until it processes once. This is why you set the `$x` variable to `1` when you build the construct as it's implied that the first run through the loop has already executed.

Switches

Switches enable you to quickly test multiple scripting scenarios without actually writing `if` statements with comparison operators. Switches are the most efficient flow control commands as they can quickly funnel data into different code sections based on an item. The `Switch` command allows you to evaluate the contents of a single variable and execute subsequent tasks based on the value of the variable. Switches also have a `default` value that is used by the switch when none of the values equal any of the suggested values in the switch statement. To invoke the `Switch` command, you declare `Switch ($variableToEvaluate)`. The second part of the `Switch` command is to declare potential values that the `$variableToEvaluate` could be, as shown here:

```
$x = "that"
Switch ($x) {
    this { write-host "Value $x equals this." }
    that { write-host "Value $x equals that." }
    Default { write-host "Value Doesn't Match Any Other Value" }
}
```

The output of this is shown in the following screenshot:

```
PS C:\> $x = "that"
PS C:\> Switch ($x) {
>> this { write-host "Value $x equals this." }
>> that { write-host "Value $x equals that." }
>> Default { write-host "Value Doesn't Match Any Other Value" }
>>
>>
Value that equals that.
```

The preceding script displays the proper construct of a `Switch` command. This example starts by setting the `$x` variable to `that`. It then enters the `Switch` construct that compares the `$x` variable to the suggested values. In this example, the `$x` variable equals `that`, after which the `Switch` will then write to the console the message `Value that equals that`. If the value of `$x` was set to `this`, it would write to the console the message `Value this equals this`. Last, if the value of `$x` is

set to anything other than this or that, it would write to the console
Value Doesn't Match Any Other Value.

Combining the use of functions, switches, and loops

There may be instances where you will want to combine the use of the different structures explained in this chapter. The example that you will create is a simple menu system that can be modified for use within your scripts. This script will prompt for your interaction and perform actions based on your response, as shown here:

```
# A Menu System for Use With This Example
Function menu-system {
    Write-host *****
    Write-Host "* Please Make A Selection Below:"
    Write-Host "*"
    Write-Host "* [1] Backup User Permissions."
    Write-host "*"
    Write-Host "* [2] Delete User Permissions."
    Write-host "*"
    Write-Host "* [3] Restore User Permissions."
    Write-host "*"
    Write-host *****
    Write-host ""
    Write-host "Please Make A Selection:"
    # Prompt for a User Input.
    $x = $host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")
    # A Switch to Evaluate User Input.
    Switch ($x.Character) {
        1 { write-host "Option 1: User Permissions Backed Up." }
        2 { write-host "Option 2: User Permissions Deleted." }
        3 { write-host "Option 3: User Permissions Restored." }
        Default {
            return $True
        }
    }
}
# A Loop Structure That will Loop Until $Restart doesn't equal true.
Do {
    $restart = Menu-system
    If ($restart -eq $True) {
        cls
        write-host "!! Invalid Selection: Please Try Again"
    }
}
```

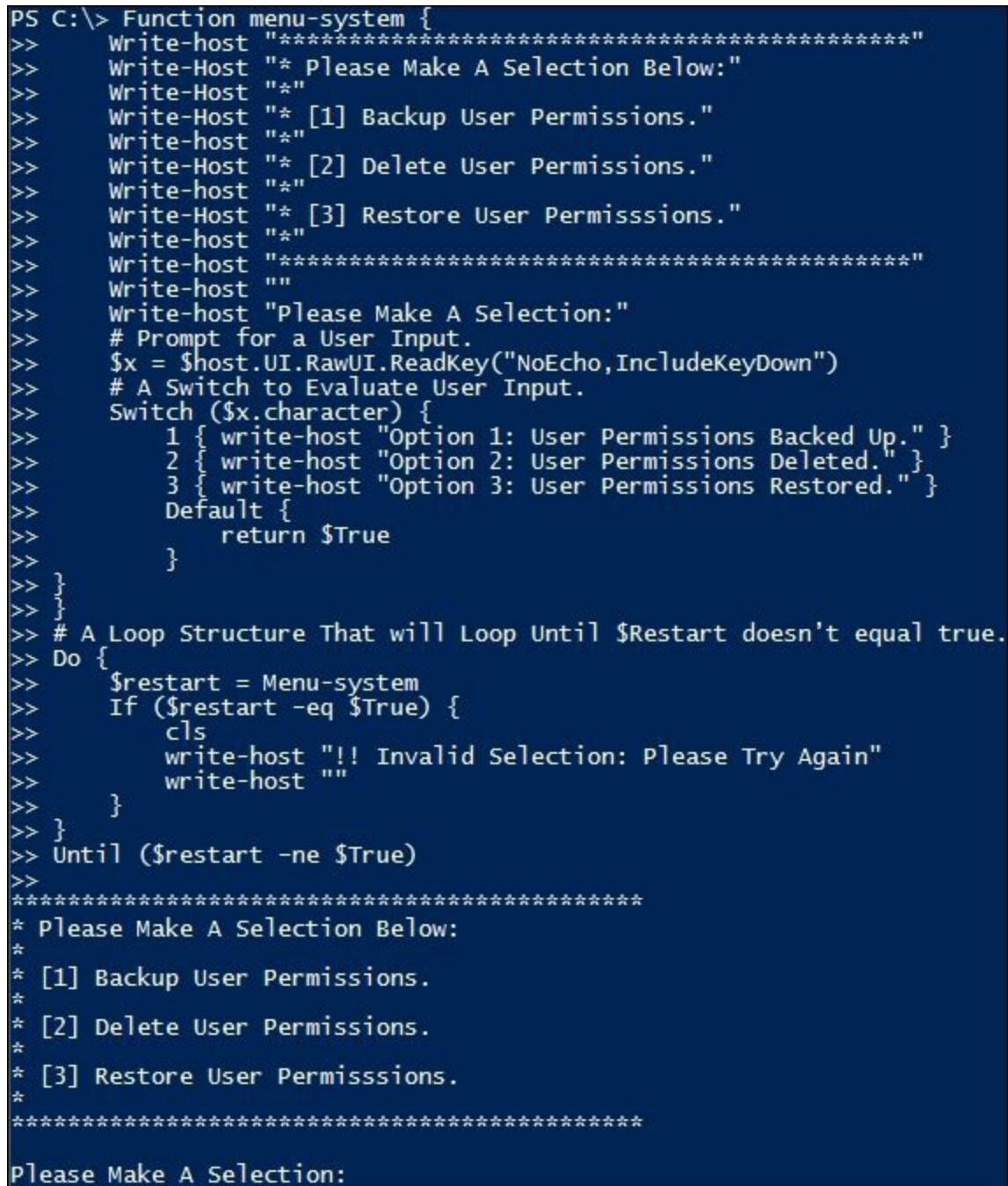
```

    write-host ""
}
}

Until ($restart -ne $True)

```

The output of this is shown in the following screenshot:



```

PS C:\> Function menu-system {
>>     Write-host "*****"
>>     Write-Host "* Please Make A Selection Below:"
>>     Write-Host "*"
>>     Write-Host "* [1] Backup User Permissions."
>>     Write-host "*"
>>     Write-Host "* [2] Delete User Permissions."
>>     Write-host "*"
>>     Write-Host "* [3] Restore User Permisssions."
>>     Write-host "*"
>>     Write-host "*****"
>>     Write-host ""
>>     Write-host "Please Make A Selection:"
>>     # Prompt for a User Input.
>>     $x = $host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")
>>     # A Switch to Evaluate User Input.
>>     Switch ($x.character) {
>>         1 { write-host "Option 1: User Permissions Backed Up." }
>>         2 { write-host "Option 2: User Permissions Deleted." }
>>         3 { write-host "Option 3: User Permissions Restored." }
>>         Default {
>>             return $True
>>         }
>>     }
>> }
>>
>> # A Loop Structure That will Loop Until $Restart doesn't equal true.
>> Do {
>>     $restart = Menu-system
>>     If ($restart -eq $True) {
>>         cls
>>         write-host "!! Invalid Selection: Please Try Again"
>>         write-host ""
>>     }
>> }
>> Until ($restart -ne $True)
>>
*****  

* Please Make A Selection Below:  

*  

* [1] Backup User Permissions.  

*  

* [2] Delete User Permissions.  

*  

* [3] Restore User Permisssions.  

*****  

Please Make A Selection:

```

This script displays the proper syntax to create a menu system within PowerShell. It first starts by declaring a function named `menu-system`. The `menu-system` function prints to the console instructions on how to

use the menu-system. It then pauses and waits for user interaction by declaring \$x = \$host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown"). When you press a key on the keyboard, the input is set to the \$x variable and the script continues. The script then enters the Switch command and evaluates the input character (\$x.Character) against the options that you set up. If you press 1, the console will write Option 1: User Permissions Backed Up. and exit the Switch. If you press 2, the console will write Option 2: User Permissions Deleted. and exit the Switch. If you press 3, the console will write Option 3: User Permissions Restored. and exit the Switch. If you press any other keys than 1, 2, or 3, the script will return \$True.

This script also leverages a Do/Until loop to restart the menu-system method each time the user presses an invalid key. The Do loop is entered and will execute the menu-system method and catches any returns from the method into the \$restart variable. Upon successful key entry from the method, the method will write to the console and exit the method. When the method exits, it doesn't return any data to the \$restart variable and the \$restart variable will be blank. Since this does not equal \$True, the Do/Until loop will successfully exit the script. Inversely, if the user doesn't enter a correct value, the Method will return \$True and set the \$restart variable to \$True. The if statement will evaluate to be \$True, clear the screen using the cls command, write to the console !! Invalid Selection: Please Try Again, write to the console a line spacer "", and restart at the top of the Do/Until loop structure.

Best practices for functions, switches, and loops

When you are scripting, you will find that you frequently need to utilize functions, loops, switches, and methods. Each of these code structures enable you to produce code faster so that you don't have to repeat code within your script. As you work with each of these structures, there are several best practices that you can follow.

Best practices for functions

There are a few recommended steps that can be followed to obtain optimum performance from functions. They are listed as follows:

- If you need to execute a sequence of code more than once, you should create a function. This will allow you to quickly repeat the same action without significantly increasing the size of the script.
- If you need to pass information into a function for processing, you should leverage arguments. Arguments will need to be declared in the order by which the function will use them.
- If you need to pass information back from a function, you should utilize the `return` command. When used with arguments, it allows you to input data, manipulate data, and return it to a variable for use in other areas of the script.
- Functions need to be declared in the script before you use them. When you are stacking multiple functions in a script, place the functions that will be used first near the top of the file, and the others can follow based on the execution order.
- When you are creating new functions, they should be named as "verb-noun". This will allow for other people to quickly read your scripts and determine what action is being performed. The most common verbs are `get-`, `set-`, `write-`, `delete-`, `read-`, `new-`, `replace-`, `insert-`, `add-`, `show-`, and `remove-`.

Best practices for looping structures and

switches

As you are working with looping structures and switches, there are several recommended best practices that will ensure scripting success, as shown here:

- It is recommended to keep the looping structures positive in nature. Use `Do/While` and `Do/Until` with the `-eq` conditional operator. This will promote performing actions `until` a variable equals a value or performing an action `while` a variable equals a value. Positive conditional operators make reading the script much easier and avoid double negative statements.
- While the `For` looping structure works well for iterative processing of multiple values, it is recommended to leverage the `ForEach` looping structure. While both looping structures will achieve the same output, `ForEach` has a much easier format to read.
- When you are declaring variables for use with the `ForEach` looping structure, it is one of the best practices to use words as variables. For example, you can declare `ForEach ($user in $list)`. This makes it clear that you want to process each `$user` in the `$list`. This is much better than stating `ForEach ($x in $y)` from a legibility standpoint.
- When you need to create multiple `if` statements in your script to evaluate a variable, you should leverage the use of switches.
- When you declare the multiple switch options, it is important to create only the necessary values that require action, and set the `default` value for all other values. This will reduce the complexity of your switch statements.

Summary

This chapter explored some of the fundamental components that are required for creating PowerShell scripts. These components include functions, loops, and switches. Each of these structures has a purpose within your scripts and can reduce the amount of effort in creating your scripts.

During this chapter, you explored how to create the structure of functions. You also learned how to feed arguments into these scripts and return values from a function.

The chapter also explained how you can create different types of looping structures. These looping structures include `Do/Until`, `Do/While`, `ForEach`, and `For`. The `Do/Until` loop was designed to execute until a variable equals a value. The `Do/While` loop will execute while a variable equals a value. The `ForEach` loop will execute for each object in an array. The final looping structure is `For`, which will execute for a set number of times as defined in the initial structure of the loop.

You went through the process of creating a `Switch`. You learned that switches are used in place of multiple "if" statements to evaluate what the contents of a variable are. You also learned that switches have a *default* value; if a switch doesn't match any of the criteria, it will execute the default section of code.

After learning about the core fundamentals of these components, we pulled the chapter together with an example on how to leverage functions, looping structures, and switches together for creating a simple menu system. This chapter ends by explaining multiple best practices that can be leveraged for the use of functions, loops, and switches.

The next chapter explores regular expressions. Regular expressions enable you to validate data syntax and structure. Regular expressions are frequently used with comparison operators, functions, loops, and switches to do advanced validation of data. You will learn how to

leverage regular expressions within your PowerShell scripts.

Chapter 5. Regular Expressions

When you are scripting, you will run into situations where you need to validate strings to see if they meet certain criteria. While you have the ability to use comparison operators to view if a string matches a certain value, you don't have the ability to validate parts of a string. A good example is of *IP addresses*. The comparison operators don't have the ability to validate the syntax of an IP address. If you were to use the normal comparison operators to validate an IP address syntax, you would have to build a script that would split the numerical values, verify that it has 4 octets, validate the individual numerical values, and pass a `True` or `False` value. Regular expressions solve this problem by providing deep comparison operations in a single string to verify that strings meet certain criteria.

In this chapter, you will learn about:

- Common metacharacters
- Grouping constructs and ranges
- Regular expression quantifiers
- Regular expression anchors
- Regular expression examples

Regular expressions are mostly language neutral. This means expressions created in a different programming language have the versatility to be used in your PowerShell scripts. While there are some minor differences between the implementations in programming languages, the base syntax is the same. With this being stated, regular expressions are not for every developer. Since there is a fairly large learning curve to using regular expressions, sometimes writing out longer evaluation scripts using comparison operators may be simpler than leveraging regular expressions. If you choose to use regular expressions in your scripts, it is recommended that you thoroughly comment the code to help any other developers quickly decipher the regular expression you are using.

Getting started with regular expressions

In the most basic form, regular expressions are used to match a specific set of characters contained in a string. To use regular expressions in PowerShell, you need to leverage the `-match`, `-notmatch`, and `-replace` operators in conjunction with an expression. The proper syntax for the comparison operators is referencing a variable or string, followed by the `-match`, `-notmatch`, or `-replace` operator and then the expression you want to evaluate. If the comparison operator is `-match` or `-notmatch`, the expression will return either `True` or `False`. If you use the `-replace` operator, the expression will return the string or variable with the replaced values.

Tip

By default, PowerShell's `-match` operator is case-insensitive. This means it will return `$True` if the letter exists. To fully leverage the regular expression's case sensitivity, use `-cmatch`. The `-cmatch` operator is for case match and will make the matching case-sensitive.

Regular expressions have characters that are reserved for use in evaluation. Each of these metacharacters have a specific meaning for the interpretation of regular expressions. Some of these common characters include:

- `\`: This character indicates an escape character. This is used to perform a literal interpretation of symbol characters. So, if you don't want the regular expression to evaluate the symbol metacharacter, place `\` in front of the character and it will use the literal character. For all other word and number characters, if you want to use the special meaning, you need to leverage the `\` symbol, and the expression will use the special meaning as seen in the following common characters.
- `\d`: This character matches a single digit. It will return `True` if the string contains a number. It will return `False` if the string contains no

numbers.

- \d: This is a negative character class of \d. It will return `True` if the string doesn't contain any character other than just numbers. It will return `False` if the string contains just numbers.
- \s: This character matches any white space character such as a space or a tab. It will return `True` if the string contains white space characters. It will return `False` if the string contains no white space characters.
- \S: This is a negative character class of \s. It will return `True` if the string doesn't contain white space characters. It will return `False` if the string contains white space characters.
- \w: This character matches any character that can be used in a word. It will return `True` if the string contains letters and numbers. It will return `False` if the string only contains symbols.
- \W: This is a negative character class of \w. It will return `True` if the string contains symbols. It will return `False` if the string only contains letters and numbers.
- .: This metacharacter is a wildcard character which indicates that it can be matched to a character. This character is commonly used inside a word to designate a wildcard character in the matching process. You can use the regular expression `Jo.`, which would match the words `Joe` and `Jon` in a string and return `True`.

To match single items in a string, do the following action:

```
# Match any character in string
"This Matches Any Character" -cmatch "."
# Match any character in string that is $null
"" -cmatch "."
# Match the Period in string
"This Matches Just The Period." -cmatch "\."
# Match the period - no periods exist.
"This Matches Nothing" -cmatch "\."
```

The output of this is shown in the following screenshot:

```
PS C:\> # Match any character in string
PS C:\> "This Matches Any Character" -cmatch "."
True
PS C:\> # Match any character in string that is $null
PS C:\> "" -cmatch "."
False
PS C:\> # Match the Period in string
PS C:\> "This Matches Just The Period." -cmatch "\."
True
PS C:\> # Match the period - no periods exist.
PS C:\> "This Matches Nothing" -cmatch "\."
False
```

This example displays how to use a regular expression to detect characters in a string. It also shows you how to properly use the escape character \ to evaluate the literal character meaning of .. You first start by declaring the string This Matches Any Character, calling the -cmatch comparison operator and the value you want to search for which is .. When PowerShell evaluates this regular expression, it will return True because the string This Matches Any Character contains a character.

The second part of this script is the evaluation of a string that contains no characters. You first declare "" followed by the -cmatch comparison operator and .. When PowerShell evaluates this regular expression, it determines that there are no characters in the string and will return False.

The third part of this script leverages the escape character. In this example, you examine the string This Matches Just The Period. by using the -cmatch comparison operator and the regular expression \.. When PowerShell evaluates this string, it searches for the character period. Since the string contains a period, the regular expression will return True.

The last part of this script also leverages the escape character. In this example, you examine the string This Matches Nothing with the -cmatch comparison operator and the regular expression of \.. Since there are no periods in the string, the regular expression returns False.

To use the `-replace` operator with regular expressions, do the following action:

```
"This is PowerShell." -replace "Power","a Turtle"
```

The output of this is shown in the following screenshot:

```
|PS C:\> "This is PowerShell." -replace "Power","a Turtle"
|This is a TurtleShell.
```

The preceding example displays how to use a regular expression to replace characters in a string. You first start by declaring the string `This is PowerShell.` calling the `-replace` comparison operator, the value you want to replace, which is `Power` followed by a comma and then the value you want to replace it with, which is `a Turtle`. When the PowerShell evaluates this regular expression, it will return `This is a TurtleShell.` because the string `This is PowerShell.` contains `Power` and replaces that value with `a Turtle`. The `-replace` operator will return `This is a TurtleShell..`

To match specific numbers, words, and nonword characters, perform the following actions:

```
# Match to Digit Characters
"0132465789" -cmatch "\d"
# Match to Non-Digit Characters
"This String Contains Word Characters" -cmatch "\D"
# Match to Word Characters
"This String Contains Words" -cmatch "\w"
# Match to Non-word Characters
"!!@#@##$" -cmatch "\W"
# Match to Space Characters
"This String Contains A Space" -cmatch "\s"
# Match to Non Space Characters
"ThisCannotContainSpaces" -cmatch "\S"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Match to Digit Characters
PS C:\> "0132465789" -cmatch "\d"
True
PS C:\> # Match to Non-Digit Characters
PS C:\> "This String Contains Word Characters" -cmatch "\D"
True
PS C:\> # Match to Word Characters
PS C:\> "This String Contains Words" -cmatch "\w"
True
PS C:\> # Match to Non-word Characters
PS C:\> "!!@#@##$" -cmatch "\W"
True
PS C:\> # Match to Space Characters
PS C:\> "This String Contains A Space" -cmatch "\s"
True
PS C:\> # Match to Non Space Characters
PS C:\> "ThisCannotContainSpaces" -cmatch "\S"
True
```

This example displays how to match a variety of data types in strings using regular expressions. The first evaluation uses the `-cmatch` comparison operator on the string `0132465789` to see if it matches the regular expression of `\d` or contains digits. As the string contains numerical characters, the regular expression returns `True`.

The second evaluation uses the `-cmatch` comparison operator on the string `This String Contains Word Characters` to see if it matches the regular expression of `\D` or contains non digits. As the string contains all word characters, the regular expression returns `True`.

The third evaluation uses the `-cmatch` comparison operator on the string `This String Contains Words` to see if it matches the regular expression of `\w` or contains word characters. As the string contains word characters, the regular expression returns `True`.

The fourth evaluation uses the `-cmatch` comparison operator on the string `!!@#@##$` to see if it matches the regular expression of `\W` or contains non-word characters. As the string contains symbols, the regular expression returns `True`.

The fifth evaluation uses the `-cmatch` comparison operator on the string `This String Contains A Space` to see if it matches the regular expression of `\s`. As the string contains space characters, the regular expression returns `True`.

space characters, the regular expression returns True.

The last evaluation uses the `-cmatch` comparison operator on the string `ThisCannotContainSpaces` to see if it matches the regular expression of `\s` or contain only whitespace characters. As the string doesn't contain only whitespace characters, the regular expression returns True.

Regular expression grouping constructs and ranges

A regular expression grouping construct is similar to what a parenthetical statement is to math operations. Group constructs bind expressions together to evaluate specific information in a specific order. The bracket [] grouping construct groups evaluation criteria together for evaluation. The regular expression will consider all data in the group [] for matching to remain `True`.

The parentheses grouping constructs in regular expressions are used to group commands together to determine the order of processing. Similar to other programming languages, equations in the parentheses are evaluated first before the rest of the expression. The parentheses grouping constructs can also be used with the OR alternation construct. The content will be evaluated as this OR that. The () operator is used with the pipe | to designates multiple OR operations. The proper syntax would be `(this|that)` to designate this OR that.

Regular expression ranges are a way to designate evaluations between two word or number characters. Ranges are used with the grouping constructs to evaluate as the expressions are more than one. To declare a range, you first start with a letter or number followed by the use of a hyphen and then end with a letter or number. When you are declaring a range, it is important to know that you need to group all lowercase, all uppercase, and all numbers separate from each other. These ranges would look like `[a-z]`, `[A-Z]`, or `[0-9]`. You don't have to use the full alphabet or number range while declaring the range. It can be a partial range.

Some examples of how grouping constructs can be used are shown here:

- `[a-z]` or `[A-Z]`: This indicates a character range from one character to another character. This will return `True` if any of the characters in the string contain the characters within the range. This will return

`False` if all of the characters are out of the range of characters provided.

- `[0-9]`: This indicates a number range from one number to another number. This will return `True` if any of the numbers in the string contain the numbers in the range. This will return `False` if all of the numbers are out of the range of numbers provided.
- `[abcd1234]`: This indicates specific characters `abcd123` that are to be matched in a string. This will return `True` if any of the `abcd1234` characters in the string contain the characters specified in the regular expression group. This will return `False` if the complete string doesn't contain the `abcd1234` characters provided in the regular expression group.
- `|`: This metacharacter indicates the alternation operator of OR. When you use this metacharacter, it will match one set of characters or another set of characters in a string. If any of the characters exist, the regular expression will return `True`.

When you are creating regular expressions, there are often times when you want to provide flexibility in your ranges. With regular expressions, you have the ability to evaluate multiple ranges such as searching for both upper and lower case word characters. Regular expressions allow you to combine the ranges together by specifying one range after another. The syntax for this would look like `[a-zA-Z1-9]`. If any of those word characters are found in upper or lower case, the evaluation would return `True`. When the expression doesn't match any of those ranges, the expression would return `False`.

To match using the OR operator and ranges, do the following action:

```
# Match Uppercase O OR Lowercase u
"Domain\User23" -cmatch "(O|u)"
# Match Uppercase O OR Uppercase U
"Domain\User23" -cmatch "(O|U)"
# Match Lowercase a-u or Uppercase A-U
"Domain\User23" -cmatch "[a-uA-U]"
# Match Lowercase a-u or Uppercase A-U or numbers 2-3
"Domain\User23" -cmatch "[a-uA-U2-3]"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Match Uppercase O OR Lowercase u
PS C:\> "Domain\User23" -cmatch "(O|u)"
False
PS C:\> # Match Uppercase O OR Uppercase U
PS C:\> "Domain\User23" -cmatch "(O|U)"
True
PS C:\> # Match Lowercase a-u or Uppercase A-U
PS C:\> "Domain\User23" -cmatch "[a-zA-U]"
True
PS C:\> # Match Lowercase a-u or Uppercase A-U or numbers 2-3
PS C:\> "Domain\User23" -cmatch "[a-zA-U2-3]"
True
```

This example displays how to use ranges and the OR alternation construct to evaluate a string. Your first expression evaluates Domain\User23 string with the `-cmatch` comparison operator and the regular expression of `(O|u)`. As the string doesn't contain an uppercase `O` or a lowercase `u`, the expression evaluates to be `False`.

The second expression evaluates Domain\User23 string with the `-cmatch` comparison operator and the regular expression of `(O|U)`. Even though the expression doesn't contain an uppercase `O`, it contains an uppercase `U`. As one of the two evaluations are `True`, the whole expression evaluates to be `True`.

The third expression evaluates Domain\User23 string with the `-cmatch` comparison operator and the regular expression range of `[a-zA-U]`. As regular expressions only require one character to be matched for the expression to be `True`, the first character evaluated in the ranges makes the whole expression to be evaluated as `True`.

The fourth expression evaluates Domain\User23 string with the `-cmatch` comparison operator and the regular expression range of `[a-zA-U2-3]`. As regular expressions only require one character to be matched for the expression to be `True`, the first character evaluated in the ranges makes the whole expression to be evaluated as `True`.

Regular expression quantifiers

When you are writing regular expressions, there are instances where you need to validate if one or more characters exist in the string being evaluated. Regular expression quantifiers evaluate a string to determine if it has a certain number of characters. In the instance of the string `ABC`, you can write a quantifier expression to evaluate that the string has at least one `A`, one `B`, one `C`, and no `D`. If the expression has the designated number of characters, it will evaluate as `True`. If the expression contains less or more than the designated amount, it will return as `False`.

The regular expression quantifiers include the following characters:

- `*`: This character requires zero or more matches of the preceding character to be `True`. This means that if you specify `abc*d`, it will match `a`, `b` and then zero or more of `c` followed by the letter `d`. In the instance of `aaabbbbccccc`, the string will evaluate to be `True` because the letters `a`, `b`, and `c` are in the exact order before the letter `d`. If the string doesn't contain `a`, `b`, then zero or more of `c` in order followed by the letter `d` somewhere in the string, it will evaluate as `False`.
- `+`: This character designates one or more matches of the referenced character. For example, if you specify the expression of `c+`, it will evaluate the string `abc` and determine that there are one or more `c` characters in the string. If the string contains one or more `c` characters, the expression would be `True`. If you evaluate `c+` against the string of `abd`, it will evaluate to be `False` as the letter `c` is not contained with in that string.
- `?`: This character designates zero or one character match. Consider this as an optional character for evaluation. This means that if you specify the expression `abc?d`, the `c` doesn't have to exist for the `abd` string to be `True`. However, if the string which is being evaluated is `cba`, it will return `False` because the prerequisite of `a` and `b` followed by `d` in order are not matched.
- `{ }`: This bracket grouping specifies a specific number or a specific range of consecutive matches that must occur in an expression for it

to be True. The proper syntax of a specific number of evaluated characters looks like `character{number}`. In the example of `a{4}`, it designates that a string must contain 4 `a` characters to evaluate to be True. The proper syntax for a range of required characters looks like `character{min,max}`. In the example of `a{3,4}`, it designates that a string must contain a minimum of 3 consecutive `a` characters and a maximum of 4 consecutive `a` characters to evaluate to be True. If the string doesn't have a minimum of 3 consecutive `a` characters and a maximum of 4 consecutive `a` characters, the expression will evaluate as False.

When you are grouping multiple expressions together, you may need to use the `.*` regular expression. This designates that any `.` character may or may not exist before or after where that expression is declared. Essentially, you are creating a *wildcard* to construct the string you are looking to evaluate. So if you were to use `.*abc.*` as the expression, it would allow any characters before and after `abc` as long as `abc` exists with in the string.

To match specific items in a string, do the following action:

```
# Match the Word "Domain" and a backslash
"Domain\User23" -cmatch "Domain.*\\.*"
# Match the Word "Domain" and a backslash
"Domain.com\User23" -cmatch "Domain.*\\.*"
# Match the Word "Domain" and a backslash
>User23.Domain.com" -cmatch "Domain.*\\.*"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Match the Word "Domain" and a backslash
PS C:\> "Domain\User23" -cmatch "Domain.*\\.*"
True
PS C:\> # Match the Word "Domain" and a backslash
PS C:\> "Domain.com\User23" -cmatch "Domain.*\\.*"
True
PS C:\> # Match the Word "Domain" and a backslash
PS C:\> "User23.Domain.com" -cmatch "Domain.*\\.*"
False
```

This example displays how to properly use the `*` metacharacter in an

expression to evaluate the syntax of a username. The first expression leverages the `-cmatch` comparison operator to evaluate if `Domain\User23` matches the expression of `Domain.**\.*`. When you break down the expression, it evaluates if the string starts with `Domain` followed by any set of characters and then the mandatory escaped character of `\` which is again ending with any set of characters. When the expression evaluates the string as `Domain \ User23`, it will return `True` as it follows the expression pattern.

The second expression leverages the `-cmatch` comparison operator to evaluate if `Domain.com\User23` matches the expression of `Domain.**\.*`. When you break down the expression, it evaluates whether the string starts with `Domain` followed by any set of characters and then the mandatory escaped character of `\` which again ends with any set of characters. When the expression evaluates the string as `Domain .com \ User23`, it will return `True` because it follows the expression pattern.

The last expression leverages the `-cmatch` comparison operator to evaluate whether `User23.Domain.com` matches the expression of `Domain.**\.*`. When you break down the expression, it evaluates whether the string starts with `Domain` followed by any set of characters and then the mandatory escaped character of `\` which again ends with any set of characters. When the expression evaluates the string as `User23.Domain.com`, it will return `False` because it doesn't contain the right order of the evaluation criteria.

To match at least one sequence in a string, do the following action:

```
# Match the Word "Domain" at least once.  
"Domain\User23" -cmatch "Domain+"  
# Match the Word ".com" at least once.  
"Domain\User23" -cmatch "\.com+"  
# Match the Word "Domain.com" at least once and a backslash  
"Domain.com\User23" -cmatch "Domain\.com+.*\\\"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Match the word "Domain" at least once.  
PS C:\> "Domain\User23" -cmatch "Domain+"  
True  
PS C:\> # Match the word ".com" at least once.  
PS C:\> "Domain\User23" -cmatch "\.com+"  
False  
PS C:\> # Match the word "Domain.com" at least once and a backslash  
PS C:\> "Domain.com\User23" -cmatch "Domain\.com+.*\\\""  
True
```

This example displays how to use the + metacharacter to evaluate the syntax of a username. The first expression leverages the `-cmatch` comparison operator to evaluate if `Domain\User23` matches the expression of `Domain+`. When you break down the expression, it evaluates whether the string contains one or more `Domain` strings. When the expression evaluates the string as `Domain\User23`, it will return `True` as the string contains one or more instances of the string `Domain`.

The second expression leverages the `-cmatch` comparison operator to evaluate if `Domain\User23` matches the expression of `\.com+`. When you break down the expression, it evaluates whether the string contains one or more `.com` strings. When the expression evaluates the string as `Domain\User23`, it will return `False` because the string doesn't contain `.com`.

The last expression leverages the `-cmatch` comparison operator to evaluate whether `Domain.com\User23` matches the expression of `Domain\.com+.*\\\"`. When you break down the expression, it evaluates whether the string contains one or more `Domain.com` strings and at least one backslash in the evaluated string. When the expression evaluates the string as `Domain.com\User23`, it will return `True` as the string contains one or more instances of the string `Domain.com` and one backslash.

To evaluate optional characters exist, do the following action:

```
# Match "Domain", optional "com", and a backslash  
"Domain.com\User23" -cmatch "Domain.*(com)?\\\""  
# Match "Domain", optional "com", and a backslash  
"Domain\User23" -cmatch "Domain.*(com)?\\\""  
# Match "Domain", optional "com", and a backslash
```

```
"Domain.comUser23" -cmatch "Domain.*(com)?\\\"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Match "Domain", optional "com", and a backslash
PS C:\> "Domain.com\User23" -cmatch "Domain.*(com)?\\\"
True
PS C:\> # Match "Domain", optional "com", and a backslash
PS C:\> "Domain\User23" -cmatch "Domain.*(com)?\\\"
True
PS C:\> # Match "Domain", optional "com", and a backslash
PS C:\> "Domain.comUser23" -cmatch "Domain.*(com)?\\\"
False
```

This example displays how to use the ? metacharacter to evaluate the syntax of a username. The first expression leverages the `-cmatch` comparison operator to evaluate if `Domain.com\User23` matches the expression of `Domain.*(com)?\\\"`. When you break down the expression, it evaluates whether the string contains the `Domain` string followed by any characters, the optional word of `com`, and the `\` character. When the expression evaluates the string as `Domain.com\User23`, it will return `True`. This is a result of the string containing the `Domain` string, the optional word of `com`, and the `\` character.

The second expression leverages the `-cmatch` comparison operator to evaluate whether `Domain\User23` matches the expression of `Domain.*(com)?\\\"`. When you break down the expression, it evaluates whether the string contains the `Domain` string, followed by any characters, the optional word of `com`, and the `\` character. When the expression evaluates the string as `Domain\User23`, it will return `True`. This is a result of the string containing the `Domain` string and the `\` character. As `com` is an optional requirement, this will not cause the expression to return `False`.

The last expression leverages the `-cmatch` comparison operator to evaluate if `Domain.comUser23` matches the expression of `Domain.*(com)?\\\"`. When you break down the expression, it evaluates whether the string contains the `Domain` string followed by any characters, the optional word of `com`, and the `\` character. When the expression evaluates the string as `Domain.comUser23`, it will return `False`. This is a result of the

string missing one of the required characters of the \ character. Despite the other components existing, all conditions must return true for the expression to be True.

To verify that a string has one or more instances of something, do the following action:

```
# Match exactly one "Domain" and exactly one backslash
"Domain\User23" -cmatch "Domain{1}.*\\{1}"
# Match exactly one "Domain" and exactly two backslashes
"Domain\User23" -cmatch "Domain{1}.*\\{2}"
# Match exactly one "Domain" and exactly one backslash
"User32.Domain.com" -cmatch "Domain{1}.*\\{1}"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Match exactly one "Domain" and exactly one backslash
PS C:\> "Domain\User23" -cmatch "Domain{1}.*\\{1}"
True
PS C:\> # Match exactly one "Domain" and exactly two backslashes
PS C:\> "Domain\User23" -cmatch "Domain{1}.*\\{2}"
False
PS C:\> # Match exactly one "Domain" and exactly one backslash
PS C:\> "User32.Domain.com" -cmatch "Domain{1}.*\\{1}"
False
```

This example displays how to leverage the {} quantifier to verify the syntax of a username. The first expression leverages the -cmatch comparison operator to evaluate if Domain\User23 matches the expression of Domain{1}.*\\{1}. When you break down the expression, it evaluates whether the string contains only one instance of Domain string, followed by any characters, and only one instance of the \ character. When the expression evaluates the string as Domain\User23, it will return True. This is a result of the string containing only one Domain string and only one \ character.

The second expression leverages the -cmatch comparison operator to evaluate whether Domain\User23 matches the expression of Domain{1}.*\\{2}. When you break down the expression, it evaluates whether the string contains only one instance of Domain string, followed by any characters, and only two instances of the \ character. When the

expression evaluates the string as Domain\User23, it will return False. This is a result of the string containing only one Domain string and only one \ character.

The last expression leverages the -cmatch comparison operator to evaluate whether User23.domain.com matches the expression of Domain{1}.*\\{1}. When you break down the expression, it evaluates whether the string contains only one instance of Domain string, followed by any characters, and only one instance of the \ character. When the expression evaluates the string as User23.domain.com, it will return False. This is a result of the string containing only one Domain string and not the \ character.

Regular expression anchors

Anchors are used to tell the regular expression where to start and end the evaluation of a string. The most common anchors evaluate the characters at the beginning or the end of a string. This allows you to validate that the string starts and/or ends with a digit, symbol, or letter character. The most common anchors are:

- `^` and `\A`: The `^` and `\A` anchor characters indicates matching at the start of a string for evaluation. If you want to ensure that a certain pattern is matched at the beginning, you will use `^` or `\A`. The `\A` syntax is symbolic of the first character in the alphabet which is `a`. This is why regular expressions use this character to designate the evaluation from the *start* of a string.
- `$` and `\Z`: The `$` and `\Z` anchor characters indicate matching at the end of a string for evaluation. If you want to ensure that a certain pattern is matched at the end, you will use `$` or `\Z`. The `\Z` syntax is symbolic of the last character in the alphabet which is `z`. This is why regular expressions use this character to designate the evaluation from the *end* of a string.
- `\b`: The `\b` anchor characters indicate a whole word boundary. The `\b` character matches a whole word in a string rather than just an individual character or character type. If the whole word exists, the expression will return `True`. If the whole word doesn't exist, the expression will return `False`.
- The placement of this anchor is important. In the example of `\btest`, `\b` is placed in front of the word, which means it will match all words that begin with `test` like `testing`. If you choose the expression `test\b`, `\b` is placed at the end of the word, which means it will match all words that end with `test`, such as `contest`.
- `\B`: The `\B` anchor is a negative character class indicating a "not whole word" boundary. `\B` matches a whole word in a string rather than just an individual character or character type. If the whole word doesn't exist, the expression will return `True`. If the word partially makes up another word, the expression will also return `True`. If the whole word exists, the expression will return `False`. If

the word partially doesn't make up another word, it will also return False.

To evaluate strings using anchors, do the following action:

```
# Match at the beginning, if the string contains a word.  
"Successfully connected to Active Directory." -cmatch "^\\w"  
# Match at the end, if the string contains a word. (does not)  
"Successfully connected to Active Directory." -cmatch "\\w$"  
# Match at the end, if the string doesn't contain a word.  
"Successfully connected to Active Directory." -cmatch "\w$"  
# Match at the beginning, it contains a word, match any  
characters in between, and match at the end, it doesn't contain  
a word.  
"Successfully connected to Active Directory." -cmatch  
"^\\w.*\\w$"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Match at the beginning, if the string contains a word.  
PS C:\> "Successfully connected to Active Directory." -cmatch "^\\w"  
True  
PS C:\> # Match at the end, if the string contains a word. (does not)  
PS C:\> "Successfully connected to Active Directory." -cmatch "\\w$"  
False  
PS C:\> # Match at the end, if the string doesn't contain a word.  
PS C:\> "Successfully connected to Active Directory." -cmatch "\w$"  
True  
PS C:\> # Match at the beginning, it contains a word, match any characters in between, and match at the end, it doesn't  
contain a word.  
PS C:\> "Successfully connected to Active Directory." -cmatch "^\\w.*\\w$"  
True
```

This example displays how to successfully use the ^ and \$ metacharacters to validate the syntax at the beginning and the end of a string. The first expression leverages the -cmatch comparison operator to evaluate if Successfully connected to Active Directory. matches the expression of ^\w. When you break down the expression, it evaluates whether the start of the string contains a word character. When the expression evaluates the string as Successfully connected to Active Directory., it will return True. This is a result of the first character in the string being the word character of s.

The second expression leverages the -cmatch comparison operator to evaluate whether Successfully connected to Active Directory. matches the expression of \w\$. When you break down the expression, it

evaluates whether the end of the string contains a word character. When the expression evaluates the string as Successfully connected to Active Directory., it will return False. This is a result of the last character in the string being the non word character of ..

The third expression leverages the -cmatch comparison operator to evaluate whether Successfully connected to Active Directory. matches the expression of \w\$. When you break down the expression, it evaluates whether the end of the string contains a non word character. When the expression evaluates the string as Successfully connected to Active Directory., it will return True. This is a result of the last character in the string being the non-word character of ..

The last expression leverages the -cmatch comparison operator to evaluate whether Successfully connected to Active Directory. matches the expression of ^\w.*\w\$. When you break down the expression, it evaluates whether the beginning of the string contains a word character, followed by any set of characters, and that the end of the string contains a non-word character. When the expression evaluates the string as Successfully connected to Active Directory., it will return True. This is a result of the first character in the string being a word character of s and the last character in the string being the nonword character of ..

To evaluate whole words in strings, do the following action:

```
# Matches the whole word "to".
"Error communicating to Active Directory." -cmatch "\bto\b"
# Matches the whole word "to".
"Error communicating with Active Directory." -cmatch "\bto\b"
# Matches where the whole word "to" does not exist.
"Error communicating with Active Directory." -cmatch "\Bto\B"
# Matches where the whole word "to" does not exist.
"Error communicating with AD." -cmatch "\Bto\B"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Matches the whole word "to".
PS C:\> "Error communicating to Active Directory." -cmatch "\bto\b"
True
PS C:\> # Matches the whole word "to".
PS C:\> "Error communicating with Active Directory." -cmatch "\bto\b"
False
PS C:\> # Matches where the whole word "to" does not exist.
PS C:\> "Error communicating with Active Directory." -cmatch "\Bto\B"
True
PS C:\> # Matches where the whole word "to" does not exist.
PS C:\> "Error communicating with AD." -cmatch "\Bto\B"
False.
```

This example displays how to properly use the regular expressions of \b and \B to evaluate whole words in a string. The first expression leverages the `-cmatch` comparison operator to evaluate if `Error communicating to Active Directory.` matches the expression of `\bto\b`. When you break down the expression, it evaluates whether the string contains the whole word of `to`. When the expression evaluates the string as `Error communicating to Active Directory.`, it will return `True`. This is a result of the expression matching the whole word of `to` between `communicating` and `Active`.

The second expression leverages the `-cmatch` comparison operator to evaluate whether `Error communicating with Active Directory.` matches the expression of `\bto\b`. When you break down the expression, it evaluates whether the string contains the whole word of `to`. When the expression evaluates the string as `Error communicating with Active Directory.`, it will return `False`. This is a result of the expression not being able to match the whole word of `to`. While the word `Directory` contains the letters `to`, it is part of the whole word of `Directory` and is skipped by the expression.

The third expression leverages the `-cmatch` comparison operator to evaluate whether `Error communicating with Active Directory.` matches the expression of `\Bto\B`. When you break down the expression, it evaluates whether the string doesn't contain the whole word of `to`. When the expression evaluates the string as `Error communicating with Active Directory.`, it will return `True`. This is a result of the expression being able to match the non whole word of `to` in

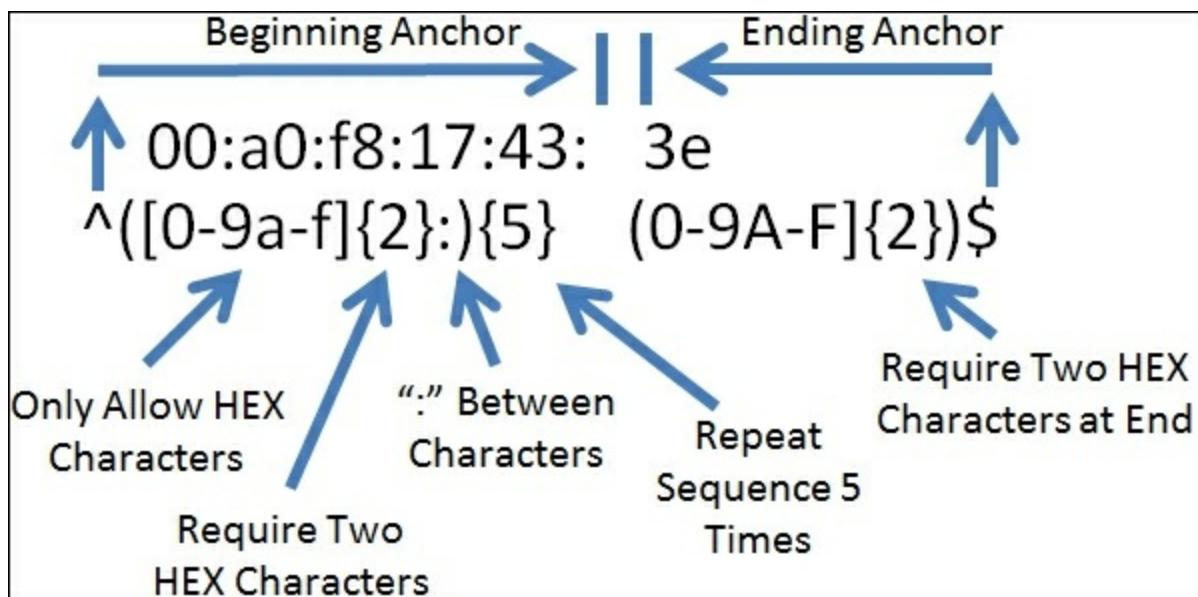
the word `Directory`.

The last expression leverages the `-cmatch` comparison operator to evaluate whether `Error` communicating with `AD.` matches the expression of `\Bto\B`. When you break down the expression, it evaluates whether the string doesn't contain the whole word of `to`. When the expression evaluates the string as `Error` communicating with `AD.`, it will return `False`. This is a result of the expression not being able to match any part of the word `to` in another word.

Regular expressions examples

When you are developing scripts, there will be many instances where you may want to use regular expressions. This section will explore regular expressions that you may run into in your environment. Since there are many methods to creating regular expressions, these are to be used as suggested starting points for your scripts.

The following diagram shows how to evaluate a MAC Address:



To test a MAC address against a regular expression, use this syntax:

```
"00:a0:f8:12:34:56" -match "^( [0-9a-f] {2} : ) {5} [0-9a-f] {2} \$"
```

The output of this is shown in the following screenshot:

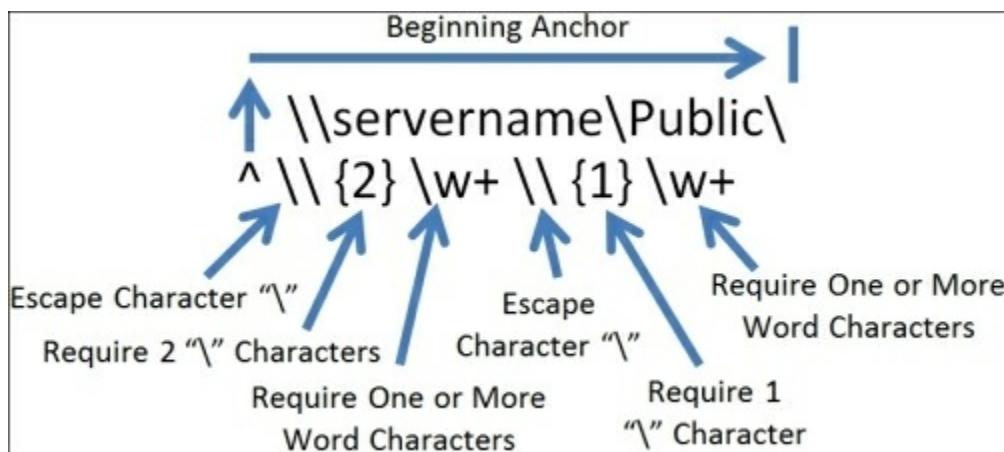
```
PS C:\> "00:a0:f8:12:34:56" -match "^( [0-9a-f] {2} : ) {5} [0-9a-f] {2} \$"
True
```

The preceding example shows how to create a regular expression to validate a MAC address. This expression leverages the `-match` comparison operator to evaluate whether `00:a0:f8:12:34:56` matches

the expression of `^([0-9a-f]{2}:){5}[0-9a-f]{2}\$`. You may choose to use the `-cmatch` operator over the `-match` operator as some applications require MAC addresses to be in uppercase. When you break down the expression, it starts by using the anchor of `^` to start evaluating from the beginning. It then uses the grouping construct of `()` to group the expression of `[0-9a-f]{2}:`.

This expression validates to see whether each character is a valid hexadecimal `[0-9a-f]` value and uses a quantifier to specify only 2 characters per sequence. The expression ends with `:`, which is the separator between each set of values. Proceeding forward, you then use another quantifier of `{5}` to repeat the two hexadecimal characters with a colon at the end 5 times. The final part of the expression is matched from the ending anchor of `\$`. The ending anchor validates to see whether each character is a valid hexadecimal `[0-9a-f]` value and uses a quantifier to specify only 2 characters as the ending of the string. When the expression evaluates the string as `00:a0:f8:12:34:56`, it will return `True`. This is a result of the expression seeing 5 sets of two hexadecimal characters, individually followed by colons, and ending with two hexadecimal characters.

The following diagram shows how to validate a UNC path:



The following syntax is used to validate a UNC path:

```
"\\servername\Public" -match "^\\{2}\\w+\\{1}\\w+"
```

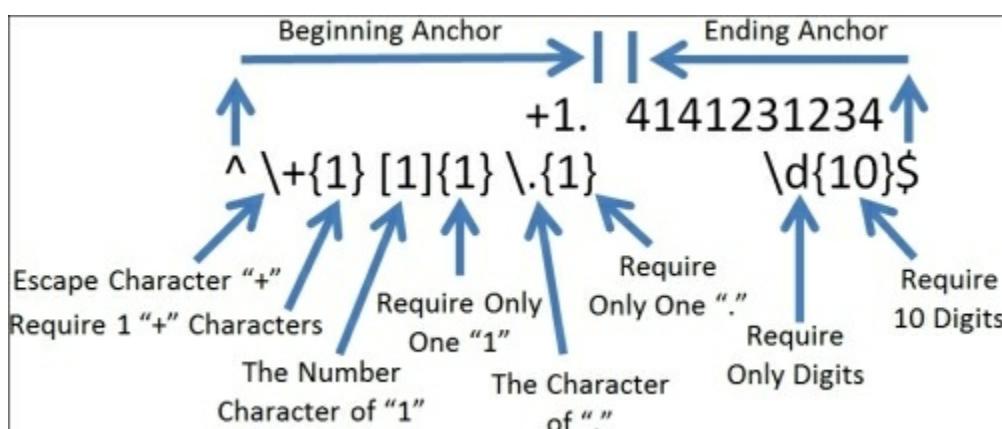
The output of this is shown in the following screenshot:

```
PS C:\> "\\\servername\Public\" -match "\\\\{2}\\w+\\\\{1}\\w+"
True
```

The preceding example displays how to validate a UNC path by leveraging regular expressions. This expression leverages the `-match` comparison operator to evaluate whether `\\\servername\Public\` matches the expression of `^\\\\{2}\\w+\\\\{1}\\w+`. You will use the `-match` operator over the `-cmatch` operator as UNC paths are not case-sensitive. When you break down the expression it starts by using the anchor of `^` to start evaluating from the beginning. It then uses the escape character of `\` to escape the backslash character. You then use a quantifier of `{2}` to require two backslashes. The expression will continue reading forward to the `\w+` expression, which will require one or more word characters. You escape the `\` character again followed by the quantifier of `{1}` to only require one backslash this time. Finally, you end the sequence with `\w+\` which requires one or more word characters to follow. When the expression evaluates the string as

`\\\servername\Public\`, it will return `True`. This is a result of the expression seeing two backslashes and then a set of word characters, followed by another backslash, and more word characters. While this doesn't validate the complete UNC path, it is a quick method to verify that the beginning of the UNC path is correct.

The following diagram shows how to create a number in the **ICANN** format:



To test a number, the following syntax is used:

```
"+1.4141231234" -cmatch "^+\{1\}[1]\{1\}\.\{1\}\d\{10\}$"
```

The output of this is shown in the following screenshot:

```
PS C:\> "+1.4141231234" -cmatch "^+\{1\}[1]\{1\}\.\{1\}\d\{10\}$"  
True
```

When the **Internet Corporation for Assigned Names and Numbers (ICANN)** changed their policy on WHOIS validation and made it mandatory from January 1, 2014, the most notable change was with the formatting of phone numbers. This example displays how to validate numbers for the ICANN standard for a United States number. This expression leverages the `-cmatch` comparison operator to evaluate whether `+1.4141231234` matches the expression of `^+\{1\}[1]\{1\}\.\{1\}\d\{10\}$`. When you break down the expression, it starts by using the anchor of `^` to start evaluating from the beginning. It then uses the escape character of `\` to escape the `+` character. You then use a quantifier of `{1}` to require one `+` characters. The expression will continue reading forward to the `[1]\{1\}` expression which uses a quantifier to require one `1` character. You escape the `.` character followed by the quantifier of `{1}` to only require one `.` character. You then use an ending character of `$` to evaluate the expression of `\d`, which requires all digits, and the quantifier of `{10}`, which makes 10 digits mandatory. When the expression evaluates the string as `+1.4141231234`, it will return `True`. This is a result of the expression seeing the character `+`, followed by number character `1`, a `.` character, and ending with 10 digits.

Summary

This chapter explained the basics of regular expressions and showed how to integrate them with PowerShell. It explained that if you are using regular expressions, you should fully comment on the code to allow other developers to read your expressions easily. This chapter further explained that PowerShell uses the `-match`, `-cmatch`, and `-replace` operators with regular expressions to evaluate criteria for strings. You also saw the most common metacharacters and how to use them in expressions. You also learned how to use grouping constructs, ranges, and qualifiers. This chapter discussed the most common anchors and concluded by providing examples of regular expressions that you may use in your scripts. These examples included a regular expression to validate a MAC address, UNC Path, and an ICANN formatted United States telephone number. In the next chapter, you will explore error and exception handling and find out about techniques for handling errors in your scripts. The next chapter will also leverage the concepts learned in the previous chapters in order to provide robust and reliable scripts.

Chapter 6. Error and Exception Handling and Testing Code

One of the most important components for creating PowerShell scripts is error and exception handling. Error and exception handling is often a forgotten component of scripting because it's common to feel that the code should always execute linearly and in an implicit fashion. While small scripts may provide low risk opportunities to not use error and exception handling, it is still recommended to use some level of error and exception handling. This is due to the common practice of taking the small scripts and using them as starting points for more complex scripts. The more complex you build your scripts, the higher the probability of failure and unexpected results.

In this chapter, you will learn the following concepts:

- Error and exception handling with parameters
- Error and exception handling with Try/Catch
- Error and exception handling with legacy exception handling
- Methodologies for testing code
- Where to test code

Utilization of strategies for testing code is an equally important component while you are developing scripts. While most developers test their code when they develop, testing the entire solution and validating the scripts are often forgotten. The more time you put into testing scenarios, the more reliable your scripts will be while they're being used in the environment. While there are many different testing strategies, any strategy is better than no strategy at all.

PowerShell has two different types of errors which are terminating and non-terminating. Terminating errors will stop the script from executing further commands. The non-terminating errors will call the `write-error` cmdlet, print an error to the screen, and continue. You will learn how to handle these different types of errors in this chapter.

Error and exception handling – parameters

PowerShell offers several different options to achieve error and exception handling. The most popular method used to catch non-terminating errors is bypassing error and exception handling parameters while executing PowerShell cmdlets. If a cmdlet detects a non-terminating error during runtime, the PowerShell **Common Language Runtime (CLR)** has the ability to store the error information in variables. You can then call the error variable and execute other actions based on the contents of the \$error variable.

The PowerShell parameters that handle error and exceptions are – `WarningAction` and `-ErrorAction`. When an issue occurs with your script, the PowerShell CLR will reference the `-ErrorAction` and `-WarningAction` arguments to determine what the next step for the script is.

There are five actions that are supported within PowerShell. The `SilentlyContinue` action will suppress the error and warning information, populate the error variables, and continue. The `Ignore` action will suppress the warning and error message and not populate any specified variables. The `Continue` action will write to the screen the warning and error information and attempt to continue with the script. The `Stop` action will write the warning and error information stop execution of the script. The `Inquire` action will prompt the end user if they want to `Halt`, `Suspend`, `Accept the Error`, or `Accept All Errors`.

The two global variables that you can set so that they provide a default error and warning action for your script are `$errorActionPreference` and `$warningActionPreference`. When you set these variables to one of the above actions, it will always default to this action for errors and warnings. By default, PowerShell is set to `Continue`, however, you can set the `$errorActionPreference` and `$warningActionPreference` to different values for different default actions.

For the warning and error actions that can place error details in a variable, the `-ErrorVariable` and `-WarningVariable` parameters can be used in conjunction with a variable name to store the error information. Proper use of these parameters would look like `-ErrorVariable err` and `-WarningVariable war`. Subsequently, the error information would be available in the `$err` variable, and the warning information would be placed in the `$war` variable.

To use cmdlet error handling, do the following action:

```
Function serviceExample($svcName) {  
    Get-service $svcName -ErrorAction SilentlyContinue -  
    ErrorVariable err  
  
    If ($err) {  
        Write-host "Error! Error Details: $err"  
        return  
    }  
  
    Write-host "Successfully Retrieved Service Information for  
$svcName. "  
}  
ServiceExample "Windows Update"  
Write-host ""  
ServiceExample "Does Not Exist"
```

The output of this is shown in the following screenshot:

```
PS C:\> Function serviceExample($svcName) {  
    >> Get-service $svcName -ErrorAction SilentlyContinue -ErrorVariable err  
    >>  
    >> If ($err) {  
    >>     Write-host "Error! Error Details: $err"  
    >>     return  
    >> }  
    >>  
    >> Write-host "Successfully Retrieved Service Information for $svcName. "  
    >> }  
    >> ServiceExample "Windows Update"  
    >> Write-host ""  
    >> ServiceExample "Does Not Exist"  
    >>  
  
    Status      Name            DisplayName  
    ----      --  
Running      wuauserv       Windows Update  
Successfully Retrieved Service Information for Windows Update.  
Error! Error Details: Cannot find any service with service name 'Does Not Exist'.
```

The preceding script displays the best method to leverage the built-in cmdlet support for error and exception handling. In this example, you create a new function named `serviceExample`. You also allow for the argument of `$svcName`. When you enter the script, you leverage the `get-service` cmdlet to query the server to see whether the data within the `$svcName` variable is a service on the system. You also pass the `-ErrorAction SilentlyContinue` to suppress messages on the screen and continue. You then specify `-ErrorVariable err` to pass any error details into the `$err` variable. If the `$err` variable has data in it or is implied true, the script will write to the console `Error! Error Details: $err` followed by `return`, which will exit out of the function. If the `$err` variable doesn't have any error details, it will proceed to write to the console `Successfully Retrieved Service Information for $svcName`. In this example, you use the `serviceExample` function to query if the Windows Update service exists. You will determine that the service does exist and the console will print the information about the service in addition to printing `Successfully Retrieved Service Information for Windows Update`.

You will then use the `serviceExample` function again to query if the Does Not Exist service exists. You will determine that the service does not exist and the script will write to the console the message `Error! Error Details: Cannot find any service with service name 'Does Not Exist'`.

Tip

When you are using `-ErrorVariable` and `-WarningVariable`, it is common to want to declare a variable name including the dollar sign (`$`). It is important to remember that you need to declare what the variable name will be but omit the dollar sign (`$`). PowerShell will create the variable anew or reuse an existing variable and then populate the error data within it.

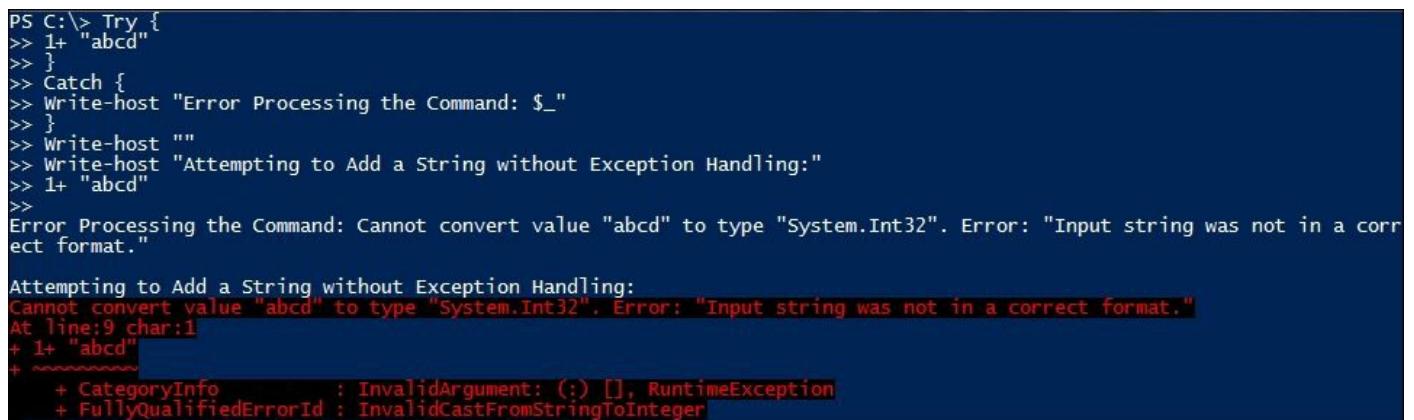
Error and exception handling – Try/Catch

One of the more popular error and exception handling techniques is leveraging Try/Catch methodology. The Try/Catch block is used for handling terminating errors and has a very simple structure. You first use the Try { } section of code and then use Catch { } to catch any errors and perform actions based on the errors. In the instance that you catch an error, you can access the exception object by declaring \$_. The \$_ refers to what is in the current pipeline. Since an error occurred during the Try sequence, the data in the pipeline is the actual error information.

To use the Try/Catch block, do the following action:

```
Try {
    1+ "abcd"
}
Catch {
    Write-host "Error Processing the Command: $_"
}
Write-host ""
Write-host "Attempting to Add a String without Exception Handling:"
1+ "abcd"
```

The output of this is shown in the following screenshot:



```
PS C:\> Try {
>> 1+ "abcd"
>>
>> }
>> Catch {
>> Write-host "Error Processing the Command: $_"
>> }
>> Write-host ""
>> Write-host "Attempting to Add a String without Exception Handling:"
>> 1+ "abcd"
>>
Error Processing the Command: Cannot convert value "abcd" to type "System.Int32". Error: "Input string was not in a correct format."
Attempting to Add a String without Exception Handling:
Cannot convert value "abcd" to type "System.Int32". Error: "Input string was not in a correct format."
At line:9 char:1
+ 1+ "abcd"
+
 + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
 + FullyQualifiedErrorMessage : InvalidCastFromStringToInteger
```

The preceding example shows how you have the ability to leverage the Try/Catch block during runtime. In this example, you enter the Try block by declaring Try { . You then attempt to add a number value to a string. Since these are two different data types, the action will throw an exception. The Catch block is then declared with Catch { . It will handle the error and write the Error Processing the Command: \$_ error to the console. \$_ is replaced with the current pipeline, which in this case is the error message. The console will read Error Processing the Command: Cannot convert value "abcd" to type "System.Int32". Error: "Input string was not in a correct format.".

In the second part of this script, you write a line separator with write-host "" and write to the console Attempting to Add a String without Exception Handling:. You then attempt to add a number value to a string outside of the Try/Catch block. When you perform this outside of the Try/Catch block, you still receive the same error details. In addition to this, you see all of the other properties associated with the exception that was thrown. From what you see on the screen, when you leverage the Try/Catch block, it will much more gracefully handle the exception.

Error and exception handling –Try/Catch with parameters

One of the best practice techniques for error and exception handling is to combine the use of the Try/Catch block and cmdlet parameters. This is due to PowerShell being able to gracefully handle terminating and non-terminating error scenarios. For instance, if you execute a line of code that throws a warning message but doesn't generate a terminating error, you can catch the warning and perform actions based on that warning. In that same instance, if the syntax of the cmdlet is incorrect, the Try/Catch block will be able to handle the terminating error due to the cmdlet throwing the exception and the catch method handling the terminating error. When you leverage both the handling techniques, you have a more robust solution to exception handling.

To use cmdlet error handling with the Try/Catch block, do the following

action:

```
Try {
    Get-process "Doesn't Exist" -ErrorAction SilentlyContinue
    -ErrorVariable err
}
Catch {
    Write-host "Try/Catch Exception Details: $_"
}
if ($err) {
    Write-host "Cmdlet Error Handling Error Details: $err"
}
```

The output of this is shown in the following screenshot:

```
PS C:\> Try {
>> Get-process "Doesn't Exist" -ErrorAction SilentlyContinue      -ErrorVariable err
>>
>> Catch {
>> Write-host "Try/Catch Exception Details: $_"
>>
>> if ($err) {
>> Write-host "Cmdlet Error Handling Error Details: $err"
>>
>> Cmdlet Error Handling Error Details: Cannot find a process with the name "Doesn't Exist". Verify the process name and ca
ll the cmdlet again.
```

The preceding example displays the proper method used to incorporate both the exception handling techniques with the `get-service` cmdlet. You first start by calling the `Try` block in use with the `get-service` cmdlet to retrieve information about a service named `Doesn't Exist`. You then tell the cmdlet to `SilentlyContinue` as `ErrorAction` and store the error details in a variable named `Err`. You then declare the `Catch` block, which will catch the output of any errors that occur during the execution of `get-service`. If an exception occurs, write to the console `Try/Catch Exception Details: $_`. You also create an `If` statement "if the `$err` variable contains data, write to the console `Cmdlet Error Handling Error Details: $err`, where `$err` contains the error information.

When you execute the script, you see that the `Catch` method doesn't catch the error message from the `get-service` cmdlet. This is due to the error being a non-terminating error, and so it doesn't invoke the `Try/Catch` block. When you run the script however, the cmdlet properly

handles the error and places the error details in the \$err variable. The script prints the message Cmdlet Error Handling Error Details: Cannot find a process with the name "Doesn't Exist". Verify the process name and call the cmdlet again. to the console.

Error and exception handling – legacy exception handling

When you are developing your scripts, you may run into command-line tools that don't have a built-in exception handling function. This is common for third-party tools that only output the debugging information to the screen, and do not properly handle the exception. One of the most common methods to handle legacy command-based applications is by the use of variables. When you execute a command, you have the ability to check the output from that command. If the output is something other than what you expect, you have the ability to throw an exception.

For example, the `netsh` command has the ability to add firewall rules to your system. When you successfully add a firewall rule to a system, the response to the console is `Ok..` By leveraging a variable, you have the ability to catch the response from this command. If the response is anything other than `Ok..`, you have the ability to write to the console the data contained in the variable.

Tip

To add a firewall rule on a system using the `netsh` command, you need to open PowerShell with the **Run as Administrator** option.

To catch a legacy command, you can perform the following action:

```
$err = netsh advfirewall firewall add rule name="Test Allow  
12345" protocol=TCP dir=out localport=12345 action=Allow  
If ($err -notlike "Ok.") {  
    Write-host "Error Processing netsh command: $err"  
}  
Write-host "Data Contained in the Variable Err is $err"
```

The output of this is shown in the following screenshot:

```

PS C:\> $err = netsh advfirewall firewall add rule name="Test Allow 12345" protocol=TCP dir=out localport=12345 action=Allow
PS C:\> If ($err -notlike "Ok.") {
>> Write-host "Error Processing netsh command: $err"
>>
>> Write-host "Data Contained in the Variable Err is $err"
>>
Data Contained in the Variable Err is Ok.

```

The preceding script shows how to properly catch the output of a firewall rule addition from a legacy command-line tool of `netsh`. You first start the script by declaring a variable which is used to capture the output from the `netsh` command. PowerShell will then process the command of `netsh advfirewall firewall add rule name="Test Allow 12345" protocol=TCP dir=out localport=12345 action=Allow`. The output of that command is set in the `$err` variable. You then create the IF statement to throw an exception if the `$err` variable is not like the output of `Ok..`. In this instance, the command is successful and the `$err` variable is set to `Ok..`. At the end of the script, you then write to the console `Data Contained in the Variable Err is $err`, where `$err` is the output from the `netsh` command. The result that is printed to the console is `Data Contained in the Variable Err is Ok..`.

To catch a legacy command, perform the following action:

```

$err = netsh advfirewall firewall add rule name="Test Allow 12345" protocol=TCP dir=out localport=1234567 action=Allow
If ($err -notlike "Ok.") {
    Write-host "Error Processing netsh command: $err"
}

```

The output of this is shown in the following screenshot:

```

PS C:\> $err = netsh advfirewall firewall add rule name="Test Allow 12345" protocol=TCP dir=out localport=1234567 action=Allow
PS C:\> If ($err -notlike "Ok.") {
>> Write-host "Error Processing netsh command: $err"
>>
>>
Error Processing netsh command: A specified port value is not valid. Usage: add rule name=<string>      dir=in|out
      action=allow|block|bypass      [program=<program path>]      [service=<service short name>|any]      [description
=<string>]      [enable=yes|no (default=yes)]      [profile=public|private|domain|any[,...]]      [localip=any|<IPv4
address>|<IPv6 address>|<subnet>|<range>|<list>]      [remoteip=any|localsubnet|dns|dhcp|wins|defaultgateway|
<IPv4 address>|<IPv6 address>|<subnet>|<range>|<list>]      [localport=0-65535|<port range>[,...]]RPC|RPC-EPMap|IPHTTPS

```

In this script, you attempt to perform a `netsh` command to add an

additional firewall rule. This time, however, you try to add a rule with an invalid port number of 1234567. When the `netsh` command processes the command of `netsh advfirewall firewall add rule name="Test Allow 12345" protocol=TCP dir=out localport=1234567 action=Allow`, it throws an exception to the console. This exception is caught in the `$err` variable. When the command evaluates the `IF` statement, it determines that `$err` is not like `Ok..`. The script will then print to the screen `Error Processing netsh command: $err`, where `$err` is an array of strings comprising the error message.

When the error message outputs to the screen, you will see that the full exception is an array of strings that is very lengthy. This is a result of the `netsh` command providing detailed help information inclusive of the error message. Fortunately, when any command-line tool outputs multiple lines of text, the `$err` variable is automatically converted to an array. This means that each line in the error message becomes a different item in an array object, and you can reference these individual lines for error information.

To print the legacy error array to the screen, perform the following action:

```
$err = netsh advfirewall firewall add rule name="Test Allow 12345" protocol=TCP dir=out localport=1234567 action=Allow
If ($err -notlike "Ok..") {
    Write-host "Array Line 0: " $err[0]
    Write-host "Array Line 1: " $err[1]
    Write-host "Array Line 2: " $err[2]
    Write-host "Array Line 3: " $err[3]
    Write-host ""
    Write-host "Error Processing netsh command:" $err[1]
}
```

The output of this is shown in the following screenshot:

```
PS C:\> $err = netsh advfirewall firewall add rule name="Test Allow 12345" protocol=TCP dir=out localport=1234567 action=Allow
PS C:\> If ($err -notlike "Ok.") {
>> Write-host "Array Line 0: " $err[0]
>> Write-host "Array Line 1: " $err[1]
>> Write-host "Array Line 2: " $err[2]
>> Write-host "Array Line 3: " $err[3]
>> Write-host ""
>> Write-host "Error Processing netsh command:" $err[1]
>> }
>>
Array Line 0:
Array Line 1: A specified port value is not valid.
Array Line 2:
Array Line 3: Usage: add rule name=<string>
Error Processing netsh command: A specified port value is not valid.
```

The preceding example provides a deeper look into how PowerShell automatically converts the `$err` variable into an array. The conversion occurs after the `netsh` command writes more than one line of exception information into the `$err` array.

You will see that the first value `$err[0]` contains blank information. The second line `$err[1]` contains useable error details with the message `A specified port value is not valid..` The third line `$err[2]` contains blank information. The fourth line that you output provides the start of the detailed help information. As the second line `$err[1]` contains the useful error information, you can print to the screen the message `Error Processing netsh command:" $err[1]`. This will write `Error Processing netsh command: A specified port value is not valid.` to the console.

Methodologies for testing code

When you are creating PowerShell Scripts, it is imperative that you test your code along the way. While there are many different development standards which you can follow such as Scrum and Agile, they all have the same premise of "test often". This section will explore recommendations for testing your code as you are developing it so that you can provide more reliable scripts.

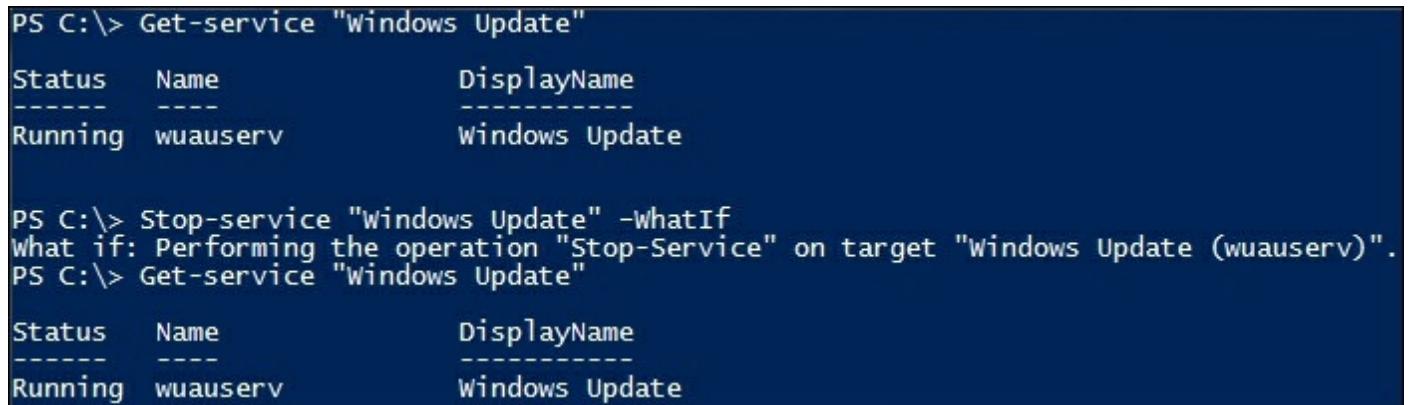
Testing the –WhatIf argument

PowerShell offers the ability to test the cmdlet's code without actually running them. This is done by the use of the `-WhatIf` argument. The `-WhatIf` argument will simulate the action that the cmdlet will take on a system without actually executing the command itself. This can assist you in determining if you have the right syntax for a command. This can also benefit you if you're getting large content from a file and need to verify that the individual items in the file won't crash your script.

To use the `-WhatIf` argument, perform the following action:

```
Get-service "Windows Update"
Stop-service "Windows Update" -WhatIf
Get-service "Windows Update"
```

The output of this is shown in the following screenshot:



The screenshot shows a PowerShell session with the following commands and output:

```
PS C:\> Get-service "Windows Update"
Status     Name            DisplayName
----     --          -----
Running   wuauserv        Windows Update

PS C:\> Stop-service "Windows Update" -WhatIf
What if: Performing the operation "Stop-Service" on target "Windows Update (wuauserv)".
PS C:\> Get-service "Windows Update"
Status     Name            DisplayName
----     --          -----
Running   wuauserv        Windows Update
```

The preceding script displays the proper use of the `-WhatIf` argument. You first start by calling the `get-service` cmdlet to retrieve information about the Windows Update Service. From running this command you determine that the Windows Update Service is currently in a running state. You then test execute the cmdlet of `stop-service` with Windows Update and the `-WhatIf` argument. Normally, the `Stop-service` command would attempt to stop the service. Instead, it outputs to the console `What if: Performing the operation "Stop-Service" on Target "Windows Update (wuaserv)".` When you call `get-service` again, you determine that the command didn't actually run; rather it successfully tested the command.

Testing the frequency

It is important to test your code as you are developing it. When you complete a small section of code, it is recommended to test that section of code independent of the entire script. The following is a list of items that should trigger the testing of your code:

- **Creation of a function or method:** When you complete a function or a method, you should always test it separately from the entire script. During scripting runtime, when you call a function, PowerShell will jump to that function or method in your script. After execution, the script will jump back to the current position in the script. The movement between the declared function and method, and the current position in the script, may cause issues with processing the subsequent code in your script. This is because the output from the functions and the methods may not be expected for subsequent steps in your script. Testing your newly created functions and methods independently and inline are essential to coding success.
- **Changes to container names in your script:** There are times where you may change the name of a variable or an array to better describe what the container is storing. When you change container names, there are instances where you might misspell or forget to update one of the containers in the full script. This is why upon changing the naming of a variable or array, you should test the script

to ensure proper execution.

- **New or updated datasets:** One of the most forgotten testing requirements is when you are leveraging a new or updated dataset such as a CSV or XML file. While your PowerShell code may not have changed, the application that generated the dataset may have values that you didn't anticipate. This could be due to the the dataset being corrupt or incomplete, or due to new values that you are not expecting in that dataset. This is why it is recommended that you test after you update a dataset.
- **Completion of a script:** When you complete a script, you should test the script in totality. While each of the individual parts of the script may execute successfully, the script may not execute in its entirety. This is why it is recommended to test the full script prior to running it on production systems.

Hit testing containers

As you are developing your scripts, it is common to create a large number of variables and arrays. As you are creating your scripts, it is also common to reuse these variables and arrays in the same script. When you are working with your scripts, it is recommended to test each of these containers to ensure that they have the data you are expecting.

To create an array and test proper formatting of the container, do the following action:

```
$array = "user.name", "joe.user", "jane.doe"  
$array
```

The output of this is shown in the following screenshot:

```
PS C:\> $array = "user.name", "joe.user", "jane.doe"  
PS C:\> $array  
user.name  
joe.user  
jane.doe
```

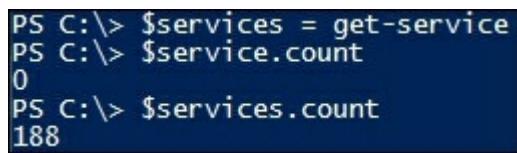
In the preceding example, you create an array \$array full of usernames

and test that the output of \$array displays the usernames properly. You should open a PowerShell window and call the \$array array to ensure the array is populated properly with usernames. This will help minimize the risk of your array being populated with erroneous information.

To view the number of services on your system, do the following action:

```
$services = get-service  
$service.count  
$services.count
```

The output of this is shown in the following screenshot:

A screenshot of a PowerShell window. The command '\$services = get-service' is entered, followed by '\$service.count', which returns '0'. Finally, '\$services.count' is entered and returns '188'.

```
PS C:\> $services = get-service  
PS C:\> $service.count  
0  
PS C:\> $services.count  
188
```

Another technique to test containers is through the use of the `count` method. When you are grabbing a large dataset from a system, you should test and count the number of items in the dataset as you are working with them. The preceding script displays why it's important to use the `count` method to test your script. You first start by declaring a variable of `$services` and by using the `get-service` cmdlet to gather all of the services on the system. You then erroneously try to count a variable named `$service` with the use of `$service.count`. The result returns `0`. Since this is obviously not the variable you used to catch the `get-service` object, you can now dig deeper into the script to determine your error. You will see that you executed the `count` method on the variable named `$services`, and it returns 188 services.

Hit testing becomes more important when you are leveraging external data sources such as CSV or XML files as the margin for error becomes greater. This is why it is recommended that you hit test and validate your variables and arrays as you go along to ensure the quality of your scripts.

Don't test in production

Testing your code on production systems is something that should never be done. The truth is, however, that many people still do test their code on production systems. Whether it is the lack of a testing platform or tight timelines, most people ignore the level of risk to complete the task at hand.

If it is absolutely necessary that you need to test your code on production systems, there are a few things that you can do to lower the overall risk:

- **Create an isolated environment to test:** On the production systems, if possible, create a segregated zone against which you can execute the PowerShell script. While you may still be leveraging production systems, you minimize the impact to the global system by segregating the script execution. For example, if you are creating a script to migrate a large quantity of users into Active Directory, create a test organizational unit into which you can populate the new users. This will minimize the risks without impacting production organizational units.
- **Clone your production system:** Virtualization provides the ability to quickly clone and create new virtual systems. If your production system is virtual, it might be possible to clone the production system and use the cloned system for testing. You also may have the ability to isolate this cloned system to a test environment to ensure that if something doesn't function properly, you don't impact the rest of your environment.
- **Leverage the -WhatIf argument:** Before you execute the commands on the production systems, try to leverage the `-WhatIf` argument. This will potentially vet out issues that will occur during the execution of your scripts.
- **Utilize write-debug:** In situations where you are running command-line tools that are not PowerShell cmdlets, you can leverage `write-debug`. If you put `write-debug` with the command-line tool parameters in quotations, you have the ability to print to the console

the full syntax for the execution of that command-line tool. This will ensure that all variables are being populated properly for execution of that command. This will help you vet out any misspelled variables and ensure the command syntax is correct.

- **Use a small dataset for testing:** If you are executing a script that uses a dataset for execution (such as a list of users), it is recommended to break off a part of that dataset and use that small section for testing. This will ensure that if something doesn't function properly during execution, you will minimize its impact on the production systems.

Summary

This chapter displays how to properly incorporate error and exception handling in your scripts. You explored how to leverage built-in cmdlet parameters for error and exception handling, and how to work with warnings and errors in your scripts. You also learned the `Try/Catch` block and how to use that to catch errors outside of what cmdlets can handle. You also learned how to use the cmdlet parameter for error and exception handling in conjunction with the `Try/Catch` block to provide robust solutions for catching issues in your scripts.

This chapter also explored methods with which you can work with legacy command-line tools and catch error messages from these tools. You learned that while the legacy tools may not have robust exception handling, PowerShell is dynamic enough to catch the errors and parse it for use in your scripts. You learned that PowerShell will automatically take multiple lines from an error message from a legacy command-line tool and place them as new items in an array.

Finally, this chapter ended with offering recommendations and methodologies for testing your PowerShell code. You learned that PowerShell has a `-WhatIf` argument which allows you to test the output from PowerShell cmdlets without actually executing them. You discovered that there are four main triggers available to test your scripts. These can be used after the creation of a function or a method, after changing container names in your script, updates in existing datasets or new datasets, and after the completion of a script. You also explored that you should always test your containers as you are building them as it will reduce typographic errors in the container names themselves. You completed this chapter by learning that you shouldn't test your code on production systems. If you need to, however, you have the ability to lower the risk by creating an isolated environment, cloning production systems, leveraging the `-WhatIf` argument, utilizing `write-host` and `write-debug`, and using small datasets for testing. In the next chapter, you will dive into the process of remotely executing PowerShell commands on systems. It will show you how to perform session-based

remote management.

Chapter 7. Session-based Remote Management

When you are developing your scripts, you may run into situations where you need to configure remote systems. While a lot of command-line programs provide the ability to execute remote commands, PowerShell provides **Common Information Model (CIM)** cmdlets allowing the scripts to be executed on remote systems over a session. The CIM cmdlet brokers the communications which provides improved performance and reliability while executing a group of commands on multiple remote systems.

Microsoft's implementation of the CIM cmdlets was derived from the need to communicate with both Windows and non Windows systems from a singular command base. Microsoft initially created **Web Services for Management (WS-Man)**, which allows communications to non Windows systems. This was problematic due to the protocol being SOAP-based and made it difficult to quickly create PowerShell scripts to communicate with these systems.

With the release of PowerShell 2.0, Microsoft developed **Windows Remote Management (WinRM)**, which created a wrapper around the WS-Man protocol known as CIM cmdlets. This greatly simplifies the communication structure and reduces the size of your scripts. CIM cmdlets communicate over port number 5985 for HTTP and 5986 for HTTPS. This is one of the biggest benefits over **Remote Procedure Call (RPC)**-based programs, due to the large quantity of upper port ranges that are required to be opened for RPC communications.

In this chapter you will learn the following concepts:

- Utilizing CIM sessions
- Creating a session
- Creating a session with session options
- Using sessions for remote management

- Removing sessions

Note

To follow the examples in Module 3, it is recommended that you start by executing the code on two to three systems that are on the same domain and same subnet and have the firewalls disabled. You will also need to start the **Windows Remote Management Service (WS-Management)** service. The commands should be executed with the administrator accounts, and all systems should have Windows Management Framework 4.0 for PowerShell 4.0. This will reduce the complexities around permissions and port configurations.

Utilizing CIM sessions

The `new-cimsession` cmdlet leverages two different protocols for communication with local and remote systems. Communications to the local system leverages the **Distributed Component Object Model (DCOM)** protocol, and remote sessions leverage the WS-Man protocol. Remote systems will need to have port 5985 for HTTP and 5986 for HTTPS opened on their firewalls to enable proper communications over WinRM. While DCOM is the default communication protocol for a local system, you can use the `- CIMSessionOptions` parameter to force WS-Man communications.

In addition to opening firewall ports, you also need to enable the WS-Management Service on the system that you are executing the commands on and the system you are remoting to as well. On most systems, this service will be set to "manual" and will not be in a running state.

The last requirement for proper communications is configuring the WinRM service. Before configuring these options, you should be aware of the security implications. When you allow all systems or a range of IPs to leverage the WinRM service, you may be putting your system at risk. Always choose options that will enable the least amount of privilege necessary to complete the task.

One method to manually configure the WinRM service includes performing the following steps:

1. Configure the Local Account Token Filter Policy in the registry.
This is done by setting the
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`
key equal to 1.
2. Enable Kerberos for use with the WinRM service. This is done with the command of:

`winrm set winrm/config/service/auth @{Kerberos="true"}`
3. Create a listener to accept requests on an IP address on the system via HTTP or HTTPS, as shown here:
 - For HTTP this is performed through this:

`winrm create winrm/config/Listener?`
`Address=*+Transport=HTTP`
 - For HTTPS this is performed through this:

`winrm create winrm/config/Listener?`
`Address=*+Transport=HTTPS @{Hostname="Host`
`Computer";CertificateThumbprint=" 40-digit hex string`
`thumbprint"}`

The syntax for both of these commands will allow WinRM communications on all IP addresses on the local system. If you want to restrict to only specific interfaces / IP addresses, replace the * with an IP address.

You may also choose to configure an Active Directory group policy that will enable the use of WinRM in your environment. The primary Windows Remote Management and Windows Remote Shell policies can be found under Windows Components and Administrative Templates in group policy.

An alternative method to configure a system for WinRM is by leveraging the `Set-WSManQuickConfig` cmdlet. This configures all of the required settings from a single command. While this may be convenient,

its down side is that it automatically configures less restrictive policies than what can be set manually. It is common for people to use a blend of the automated configuration, manual configuration, and group policy configuration.

To leverage the quick configuration, do the following action:

Set-WsManQuickConfig

The output of this is shown in the following screenshot:

```
PS C:\> Set-WsManQuickConfig
WinRM Quick Configuration
Running the Set-WsManQuickConfig command has significant security implications, as it enables remote management through the WinRM service on this computer.
This command:
1. Checks whether the WinRM service is running. If the WinRM service is not running, the service is started.
2. Sets the WinRM service startup type to automatic.
3. Creates a listener to accept requests on any IP address. By default, the transport is HTTP.
4. Enables a firewall exception for WS-Management traffic.
5. Enables Kerberos and Negotiate service authentication.
Do you want to enable remote management through the WinRM service on this computer?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): Y
WinRM is already set up to receive requests on this computer.
WinRM has been updated for remote management.
Created a WinRM listener on HTTP:///* to accept WS-Man requests to any IP on this machine.
WinRM firewall exception enabled.
Configured LocalAccountTokenFilterPolicy to grant administrative rights remotely to local users.
```

The preceding command displays the quick configuration of WinRM using a PowerShell window running as Administrator. When you enter the `Set-WsManQuickConfig` command and select `Y` as the option, the cmdlet will automatically configure the system for using WinRM. The cmdlet will check the WinRm Service and ensure that it's running, set the startup of the service to automatic, create a listener for all IP addresses assigned to the system over HTTP, enable the firewall exceptions, set up the computer for Kerberos, and negotiate service authentication.

Tip

It is important to note that while using `Set-WsManQuickConfig`, the cmdlet requires that all network interfaces should not be in *Public Mode*. The interface will need to be set to either *Work Mode* or *Home Mode*. The cmdlet will throw an error message that the WinRM Firewall exception will not work on public networks. To get around this issue,

disable the network interface that is set to "public" and try to run the command again. This is by design a security precaution to ensure that people on the public interfaces don't attempt to remotely manage your system in an unauthorized manner.

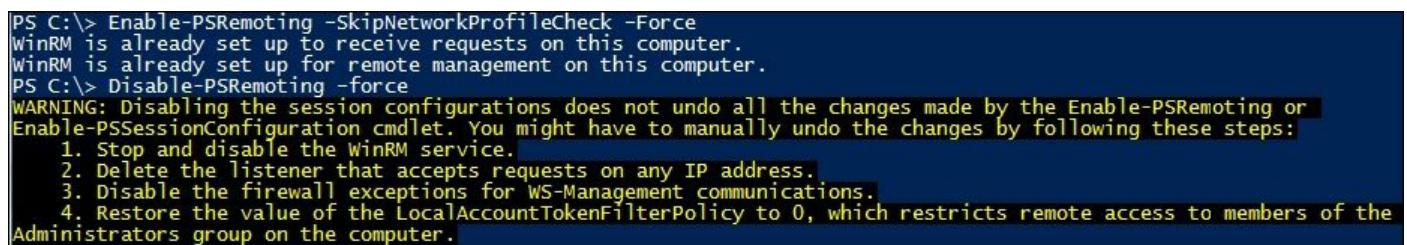
An alternative to enable WinRM on a system is to leverage the `Enable-PSRemoting` cmdlet. This cmdlet will configure the necessary components on a client machine to receive commands. There are several common parameters that can be used in conjunction with the `Enable-PSRemoting` cmdlet, which include the `-SkipNetworkProfileCheck` parameter which skips the *Public Mode* check. You may also leverage the `-force` parameter which suppresses all prompts while enabling WinRM on a system.

If you want to disable WS-Management on a system, you can leverage the `Disable-PSRemoting` cmdlet. This also accepts the `-force` parameter to suppress all prompts while disabling WinRM on a system.

To leverage the `Enable-PSRemoting` CMDlet, do the following action:

```
Enable-PSRemoting -SkipNetworkProfileCheck -Force
Disable-PSRemoting -force
```

The output of this is shown in the following screenshot:



```
PS C:\> Enable-PSRemoting -SkipNetworkProfileCheck -Force
WinRM is already set up to receive requests on this computer.
WinRM is already set up for remote management on this computer.
PS C:\> Disable-PSRemoting -force
WARNING: Disabling the session configurations does not undo all the changes made by the Enable-PSRemoting or
Enable-PSSessionConfiguration cmdlet. You might have to manually undo the changes by following these steps:
  1. Stop and disable the WinRM service.
  2. Delete the listener that accepts requests on any IP address.
  3. Disable the firewall exceptions for WS-Management communications.
  4. Restore the value of the LocalAccountTokenFilterPolicy to 0, which restricts remote access to members of the
Administrators group on the computer.
```

The preceding example displays how to enable and disable PSRemoting on a system. You first start by calling `Enable-PSRemoting` with the `-SkipNetworkProfileCheck` and `-Force` parameters. As shown in this example, the `Enable-PSRemoting` cmdlet can detect that WinRM is partially configured. In this case, it will only enable the components that are required for the service to start working properly. After executing

this command, WinRM is ready to start accepting commands over WinRM.

In the second part of this example, you leverage the `Disable-PSRemoting` cmdlet with the `-Force` trigger to disable WinRM. After executing the command, WinRM will no longer accept commands on a system.

Note

After leveraging the `Disable-PSRemoting` cmdlet , you may still need to manually disable other components of WinRM to fully disable it on a system. To do this, you can reverse the manual configuration settings described earlier, which will disable the remaining components that the `Disable-PSRemoting` cmdlet doesn't disable.

Creating a session

When you first want to establish a connection with a local or remote system, you have to create a new session. The `new-cimsession` cmdlet provides the ability to create a new session with these resources. After you create the session, all communications leveraging the CIM session to the remote systems are tunneled through this RPC session.

To create a new CIM session, do the following action:

`new-cimsession`

The output of this is shown in the following screenshot:

```
PS C:\> new-cimsession

Id      : 1
Name    : CimSession1
InstanceId : eb369bdd-9c9d-4c2f-852c-801165e86542
ComputerName : localhost
Protocol  : DCOM
```

The preceding example displays how to create a new CIM session. You

first create a session by calling just the `new-cimsession` cmdlet. You will see that the session is assigned `ID`, `Name`, and `InstanceID`. The `ID` is an incremental quantifier attribute that can be referenced to execute commands with that specific session. This typically represents the number of CIM sessions that you've created in that instance of the PowerShell command window. Subsequently, you can also reference the session by its `Name` attribute which provides a clear text way to reference a session. The `InstanceID` attribute is generated upon creation of the cmdlet. The `InstanceID` attribute is unique to only this session. The `ComputerName` attribute will reference what systems are connected to that session. The `Protocol` attribute displays that it is leveraging the `DCOM` protocol for local communications.

If you want to create a session with a specific computer to have a connection with that, you can leverage the `-computername` parameter. This allows you to create sessions to one or multiple systems separated by commas. You also have the ability to create a clear text name for the cmdlet. By leveraging the `-name` parameter, you have the ability to name the group of session in a friendly manner.

To create a new CIM session with multiple systems, do the following action:

```
new-cimsession -computername Localhost,localhost,localhost -  
name LocalSessions
```

The output of this is shown in the following screenshot:

```
PS C:\> new-cimsession -computername localhost,localhost,localhost -name LocalSessions

Id      : 1
Name    : LocalSessions
InstanceId : 1a242023-7f4d-4d65-b2c6-60731c1a5178
ComputerName : localhost
Protocol  : WSMAN

Id      : 2
Name    : LocalSessions
InstanceId : bf308272-58f3-4f4b-abde-7a73ddfeeb52
ComputerName : localhost
Protocol  : WSMAN

Id      : 3
Name    : LocalSessions
InstanceId : 86b9c4d4-046d-4ff7-ba0a-ac70a5718c85
ComputerName : localhost
Protocol  : WSMAN
```

In this example, you leverage the `new-cimsession` cmdlet to create three sessions. By leveraging the `-computername` parameter, you create three remote sessions with three computers named `localhost`. As you are calling remote computer names by utilizing the `-computername` parameter, the default protocol is `WSMAN` instead of `DCOM`. By calling the `-name` parameter, you are able to assign the group of sessions a single session name to work with the full group of sessions. You will also see that each session is assigned a unique `ID` and `InstanceId`.

Creating a session with session options

When you are creating a session, there are times where you need to configure advanced options for communication. To do this, the `new-cimsessionoption` cmdlet can be leveraged to set advanced connection options for a CIM session.

Some of these additional parameters include:

- **Protocol:** The `Protocol` parameter allows you to override the default setting for the protocol being used. This value can be either `DCOM` or `WSMAN`.
- **ProxyAuthentication:** The `ProxyAuthentication` parameter allows you to specify the authentication mechanism to the remote system. The valid values for this parameter are `Default (none)`, `Digest`,

Negotiate, Basic, Kerberos, NTLMDomain, and CredSSP.

- **ProxyCredential:** The `ProxyCredential` parameter allows you to specify credentials to authenticate to a remote system. To use the `ProxyCredential` parameter, you must access the `PSCredential` object and set this to a variable. The best way to create the `PSCredential` object is through the use of the `Get-Credential` PowerShell cmdlet. You can then use this `PSCredential` object with the `ProxyCredential` parameter to authenticate with different credentials.
- **UseSSL:** The `UseSSL` parameter forces the use of SSL for remote communications.
- **NoEncryption:** The `NoEncryption` parameter will override the default encryption values to force no encryption in communication with a remote system.

To use the `New-CimSessionOption` cmdlet, you first need to declare a variable to place the options into, followed by the options you want to create for a session. You then use the `New-CimSessionOption` variable with the `New-CimSession` command and the `-Sessionoption` parameter to create the session.

To create a new CIM session with session options, do the following action:

```
$sessionoptions = New-CimSessionOption -Protocol DCOM
New-CimSession -Sessionoption $sessionoptions -ComputerName
localhost,localhost,localhost -Name LocalSessions
```

The output of this is shown in the following screenshot:

```
PS C:\> $sessionoptions = New-CimSessionOption -Protocol DCOM
PS C:\> New-CimSession -Sessionoption $sessionoptions -ComputerName localhost,localhost,localhost -Name LocalSessions

Id      : 1
Name    : LocalSessions
InstanceId : 29c5140d-530f-45d2-9ace-131c655a1e39
ComputerName : localhost
Protocol  : DCOM

Id      : 2
Name    : LocalSessions
InstanceId : 5b35db25-f971-40ff-8f8c-15622c116f9c
ComputerName : localhost
Protocol  : DCOM

Id      : 3
Name    : LocalSessions
InstanceId : c7406bb1-24ba-4eff-9749-1cdd0bfc5d8f
ComputerName : localhost
Protocol  : DCOM
```

The preceding example displays how to properly create a CIM session with the use of the `new-cimsessionoption` cmdlet. You first start by creating a new CIM session option for forcing the communication protocol to DCOM and placing the session object to the `$sessionoptions` variable. You then use the `new-cimsession` cmdlet and the `-sessionoption` parameter to force new session options for the remote communication. You then call the `-computername` parameter to specify three remote computers and the `-name` parameter to group the three sessions into one session `Name of LocalSessions`. You will see that the output from the `new-cimsession` cmdlet will be very similar to the previous example; however, the `Protocol` is now forced to DCOM.

Using sessions for remote management

Now that you know how to create sessions, you will want to be able to leverage these newly created sessions to execute remote tasks. To be able to use a CIM session, you have to call the session by the `get-cimsession` cmdlet and then putting that session object into a variable. This is done by declaring a variable and setting it equal to the results of the `get-cimsession` command. The session object will then be contained in that variable.

To create and get a new CIM session, do the following action:

```
New-cimsession
$newsession = get-cimsession
$newsession
```

The output of this is shown in the following screenshot:

```
PS C:\> New-CimSession

Id      : 1
Name    : CimSession1
InstanceId : 7806b42d-d1c2-46fa-84ad-61c607ae750f
ComputerName : localhost
Protocol   : DCOM

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

Id      : 1
Name    : CimSession1
InstanceId : 7806b42d-d1c2-46fa-84ad-61c607ae750f
ComputerName : localhost
Protocol   : DCOM
```

This example displays how to create a new session and place that new session object into a variable. You first start by creating the new session with the `new-cimsession` cmdlet. You then call the new session through the `get-cimsession` command and by setting that session object to the variable named `$newsession`. When you call just the `$newsession` variable, you will see the session object contained in that variable.

Once you capture the session object in a variable, you have the ability to interact with that session with the use of other PowerShell cmdlets. While CIM sessions aren't supported by all of the PowerShell cmdlets, there are a variety of CIM cmdlets that do support CIM sessions. To execute a command over a CIM session, you declare the `-CIMSession` parameter with a PowerShell cmdlet and it will remotely execute this command over that session.

One of the CIM commands that you can leverage `CimSessions` with is the `Invoke-CimMethod` cmdlet. This cmdlet has the ability to invoke methods on a system like launching an application. The syntax of this command is calling `invoke-cimmethod` followed by the `-cimsession` parameter. You then call the Windows process class by using `-class win32_process`. You then create a new process by typing `-MethodName Create`. You finally issue the `-argument` parameter and issue the command-line arguments in the format of

```
@{CommandLine='programname.exe';CurrentDirectory="c:\Directory01}
```

To create a new CIM session and use the `Invoke-CimMethod` cmdlet, do the following action:

```
New-CimSession -Name MyComputer
$newsession = Get-CimSession -Name MyComputer
Invoke-CimMethod -CimSession $newsession -Class Win32_Process -Method Create -Argument
@{CommandLine='calc.exe';CurrentDirectory="c:\windows\system32"}
```

The output of this is shown in the following screenshot:

```
PS C:\> New-CimSession -Name MyComputer
Id      : 1
Name    : MyComputer
InstanceId : 5dec7441-3ebc-4342-b8c5-2eab5c2af6f3
ComputerName : localhost
Protocol : DCOM

PS C:\> $newsession = Get-CimSession -Name MyComputer
PS C:\> Invoke-CimMethod -CimSession $newsession -Class Win32_Process -MethodName Create -Argument @{CommandLine='calc.exe';CurrentDirectory="c:\windows\system32"}
      ProcessId      ReturnValue PSComputerName
      -----      -----
          5568            0          MyComputer
```

The preceding example displays how to create a new session, place that session in an object, and invoke a new calculator instance on a remote system. You first start by declaring `New-CimSession` with the name of `MyComputer`. You then use the `Get-CimSession` cmdlet to place the CIM session object into the `$newsession` variable. You use the `Invoke-CimMethod` cmdlet with the `-CimSession` parameter referencing the `$newsession` variable. You follow this by calling the `-Class` parameter of `Win32_Process` and the `-MethodName` parameter referencing the `Create` method. Finally, you pass in the arguments for the calculator instance of `@{CommandLine='calc.exe';CurrentDirectory="c:\windows\system32"}`. This will successfully launch the calculator program on a remote system.

Removing sessions

When you are working with sessions from a singular system, it is

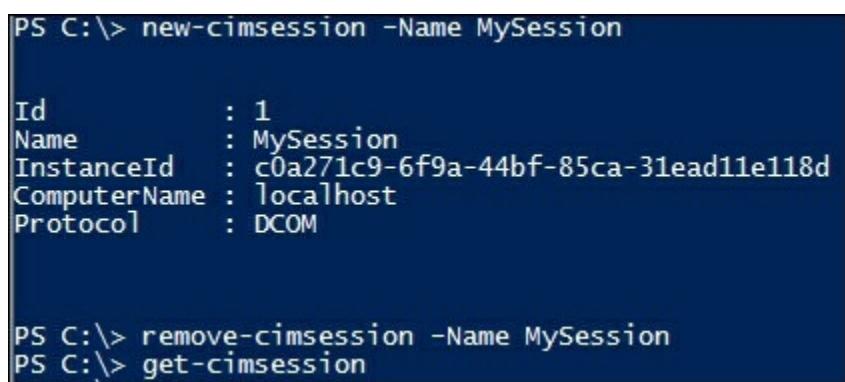
important to remember that the sessions stay alive until you close the PowerShell window or until you remove the CIM session from memory. While it may be easier to close the PowerShell window, in instances of systems that are running batch configuration jobs, you may at any point in time open thousands of CIM sessions from a singular system. This not only causes derogated performance on the originating system but also opens up opportunities for security vulnerabilities. Since it is strongly recommended to close all open sessions on a system, you can leverage the `remove-cimsession` cmdlet to close sessions after you are done with using them.

The `remove-cimsession` cmdlet can leverage any of the session identifiers to close sessions. You can use the `-Name` parameter to close sessions by `Name`, the `-ID` parameter to close by `ID`, the `-InstanceID` parameter to close by `InstanceId`, and the `-ComputerName` parameter to close by `ComputerName`.

To create and remove a CIM session, do the following action:

```
new-cimsession -Name MySession  
remove-cimsession -Name MySession  
get-cimsession
```

The output of this is shown in the following screenshot:



```
PS C:\> new-cimsession -Name MySession  
  
Id      : 1  
Name    : MySession  
InstanceId : c0a271c9-6f9a-44bf-85ca-31ead11e118d  
ComputerName : localhost  
Protocol   : DCOM  
  
PS C:\> remove-cimsession -Name MySession  
PS C:\> get-cimsession
```

In the preceding example, you use the `new-cimsession` cmdlet to create a new session with the name of `MySession`. You then run the `remove-cimession` cmdlet with the `-name` parameter to remove the session with a

name of `MySession`. Finally, you run the `get-cimsession` cmdlet to verify that there are no active sessions on the system. You will then see that after you run the `remove-cimsession` cmdlet, there are no active sessions remaining.

Summary

This chapter explained how to properly manage systems by utilizing sessions. The chapter started by explaining the prerequisites to enable the use of Windows Remote Management in your environment. This includes port numbers, security permissions, and services. The chapter also explained that you can use the quick configuration on systems, though it can be less secure than the manual configuration of WinRm.

The chapter then proceeded to explain how to create sessions through the use of the `new-cimsession` cmdlet. It also explained that you can create session options to change the connection parameters to remote systems with the `new-cimsessionoption` cmdlet. It explained that the most popular session options are `Protocol`, `ProxyAuthentication`, `ProxyCredential`, `UseSSL`, and `NoEncryption`. The chapter then highlighted that you have to set the session options object a variable prior to using them with a new CIM session.

This chapter also showed how to set a session to an object using the `get-cimsession` cmdlet. It explained how to use this session object in conjunction with the `invoke-cimmethod` to launch `calc.exe` on a remote system. The chapter ends by explaining how to close sessions with the use of `remove-cimsession` cmdlet and how to verify that all of the sessions are closed. In the next chapter, you will be exploring file, folder, and registry items with PowerShell. You will learn how to create, view, modify, and delete these items using a small set of cmdlets.

Chapter 8. Managing Files, Folders, and Registry Items

When you are automating tasks on servers and workstations, you will frequently run into situations where you need to manage files, folders, and registry items. PowerShell provides a wide variety of cmdlets that enable you to create, view, modify, and delete items on a system.

In this chapter, you will learn many techniques to interact with files, folders, and registry items. These techniques and items include:

- Registry provider
- Creating files, folders, registry keys, and registry named values
- Adding named values to registry keys
- Verifying the existence of item files, folders, and registry keys
- Renaming files, folders, registry keys, and named values
- Copying and moving files and folders
- Deleting files, folders, registry keys, and named values

Note

To properly follow the examples in this chapter, you will need to sequentially execute the examples. Each example builds on the previous examples, and some of these examples may not function properly if you do not execute the previous steps.

Registry provider

When you're working with the registry, PowerShell interprets the registry in the same way it does files and folders. In fact, the cmdlets that you use for files and folders are the same that you would use for registry items. The only difference with the registry is the way in which you call the registry path locations. When you want to reference the registry in PowerShell, you use the `[RegistryLocation] : \Path\` syntax. This is made available through the PowerShell Windows Registry Provider.

While referencing [RegistryLocation] : \Path\, PowerShell provides you with the ability to use registry abbreviations pertaining to registry path locations. Instead of referencing the full path of HKEY_LOCAL_MACHINE, you can use the abbreviation of HKLM. Some other abbreviations include:

- HKLM: Abbreviation for HKEY_LOCAL_MACHINE hive
- HKCU: Abbreviation for HKEY_CURRENT_USER hive
- HKU: Abbreviation for HKEY_USERS hive
- HKCR: Abbreviation for HKEY_CLASSES_ROOT hive
- HKCC: Abbreviation for HKEY_CURRENT_CONFIG hive

For example, if you wanted to reference the named values in the Run registry key for programs that start up on boot, the command line syntax would look like this:

```
HKLM:\Software\Microsoft\Windows\CurrentVersion\Run
```

While it is recommended that you don't use cmdlet aliases in your scripts, it is recommended, and a common practice, to use registry abbreviations in your code. This not only reduces the amount of effort to create the scripts but also makes it easier for others to read the registry locations.

Creating files, folders, and registry items with PowerShell

When you want to create a new file, folder, or registry key, you will need to leverage the `new-item` cmdlet. The syntax of this command is `new-item`, calling the `-path` argument to specify the location, calling the `-name` argument to provide a name for the item, and the `-ItemType` argument to designate whether you want a file or a directory (folder). When you are creating a file, it has an additional argument of `-value`, which allows you to prepopulate data into the file after creation. When you are creating a new registry key in PowerShell, you can omit the `-ItemType` argument as it is not needed for registry key creation.

PowerShell assumes that when you are interacting with the registry using `new-item`, you are creating registry keys. The `new-item` command accepts the `-force` argument in the instance that the file, folder, or key is being created in a space that is restricted by **User Account Control (UAC)**.

To create a new folder and registry item, do the following action:

```
New-item -path "c:\Program Files\" -name MyCustomSoftware -  
ItemType Directory  
New-item -path HKCU:\Software\MyCustomSoftware\ -force
```

The output of this is shown in the following screenshot:

```

PS C:\> New-item -path "c:\Program Files\" -name MyCustomSoftware -ItemType Directory

Directory: C:\Program Files

Mode                LastWriteTime      Length Name
----              <-----           ----- 
d----   03/16/2015   2:35 PM          0 MyCustomSoftware

PS C:\> New-item -path HKCU:\Software\MyCustomSoftware\ -force

Hive: HKEY_CURRENT_USER\Software

Name            Property
----           -----
MyCustomSoftware

```

The preceding example shows how you can create folders and registry keys for a custom application. You first create a new folder in `c:\Program Files\` named `MyCustomSoftware`. You then create a new registry key in `HKEY_CURRENT_USER:\Software\` named `MyCustomSoftware`.

You start by issuing the `new-item` cmdlet followed by the `-path` argument to designate that the new folder should be placed in `c:\Program Files\`. You then call the `-name` argument to specify the name of `MyCustomSoftware`. Finally, you tell the cmdlet that the `-ItemType` argument is `Directory`. After executing this command you will see a new folder in `c:\Progam Files\` named `MyCustomSoftware`.

You then create the new registry key by calling the `new-item` cmdlet and issuing the `-path` argument and then specifying the `HKCU:\Software\MyCustomSoftware\` key location, and you complete it with the `-force` argument to force the creation of the key. After executing this command, you will see a new registry key in `HKEY_CURRENT_USER:\Software\` named `MyCustomSoftware`.

One of the main benefits of PowerShell breaking apart the `-path`, `-name`, and `-values` arguments is that you have the ability to customize each of the values before you use them with the `new-item` cmdlet. For example,

if you want to name a log file with the date stamp, add that parameter into a string and set the -name value to a string.

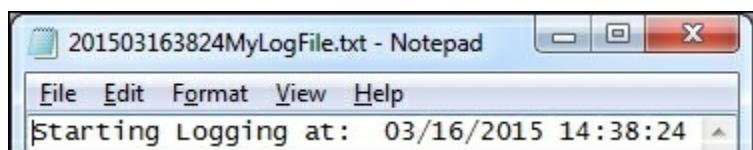
To create a log file with a date included in the filename, do the following action:

```
$logpath = "c:\Program Files\MyCustomSoftware\Logs\"  
New-item -path $logpath -ItemType Directory | out-null  
$itemname = (get-date -format "yyyyMMddmmss") + "LogFile.txt"  
$itemvalue = "Starting Logging at: " + " " + (get-date)  
New-item -path $logpath -name $itemname -ItemType File -value  
$itemvalue  
$logfile = $logpath + $itemname  
$logfile
```

The output of this is shown in the following screenshot:

```
PS C:\> $logpath = "c:\Program Files\MyCustomSoftware\Logs\"  
PS C:\> New-item -path $logpath -ItemType Directory | out-null  
PS C:\> $itemname = (get-date -format "yyyyMMddmmss") + "LogFile.txt"  
PS C:\> $itemvalue = "Starting Logging at: " + " " + (get-date)  
PS C:\> New-item -path $logpath -name $itemname -ItemType File -value $itemvalue  
  
Directory: C:\Program Files\MyCustomSoftware\Logs  
  
Mode                LastWriteTime      Length Name  
----                -----          ---- -  
-a---        03/16/2015   2:38 PM           41 201503163824LogFile.txt  
  
PS C:\> $logfile = $logpath + $itemname  
PS C:\> $logfile  
c:\Program Files\MyCustomSoftware\Logs\201503163824LogFile.txt
```

The content of the log file is shown in the following screenshot:



The preceding example displays how you can properly create a new log file with a date time path included in the log file name. It also shows

how to create a new directory for the logs. It then displays how to include text inside the log file, designating the start of a new log file. Finally, this example displays how you can save the log file name and path in a variable to use later in your scripts.

You first start by declaring the path of `c:\Program Files\MyCustomSoftware\Logs\` in the `$logpath` variable. You then use the `new-item` cmdlet to create a new folder in `c:\Program Files\MyCustomSoftware\` named `Logs`. By piping the command to `out-null`, the default output of the directory creation is silenced. You then declare the name that you want the file to be by using the `get-date` cmdlet, with the `-format` argument set to `yyyyMMddmmss`, and by adding `myfile.txt`. This will generate a date time stamp in the format of 4 digits including year, month, day, minutes, seconds, and `myfile.txt`. You then set the name of the file to the `$itemname` variable. Finally, you declare the `$itemvalue` variable which contains `Starting Log at:` and the standard PowerShell date time information. After the variables are populated, you issue the `new-item` command, the `-path` argument referencing the `$logpath` variable, the `-name` argument referencing the `$itemname` variable, the `-ItemType` referencing `File`, and the `-value` argument referencing the `$itemvalue` variable. At the end of the script, you will take the `$logpath` and `$itemname` variables to create a new variable of `$logfile`, which contains the location of the log file. As you will see from this example, after you execute the script the log file is populated with the value of `Starting Logging at: 03/16/2015 14:38:24.`

Adding named values to registry keys

When you are interacting with the registry, you typically view and edit named values or properties that are contained within the keys and subkeys. PowerShell uses several cmdlets to interact with named values. The first is the `get-itemproperty` cmdlet which allows you to retrieve the properties of a named value. The proper syntax for this cmdlet is to specify `get-itemproperty` to use the `-path` argument to specify the location in the registry, and to use the `-name` argument to specify the named value.

The second cmdlet is `new-itemproperty`, which allows you to create new named values. The proper syntax for this cmdlet is specifying `new-itemproperty`, followed by the `-path` argument and the location where you want to create the new named value. You then specify the `-name` argument and the name you want to call the named value with. Finally, you use the `-PropertyType` argument which allows you to specify what kind of registry named value you want to create. The `.PropertyType` argument can be set to `Binary`, `DWord`, `ExpandString`, `MultiString`, `String`, and `Qword`, depending on what your need for the registry value is. Finally, you specify the `-value` argument which enables you to place a value into that named value. You may also use the `-force` overload to force the creation of the key in the instance that the key may be restricted by UAC.

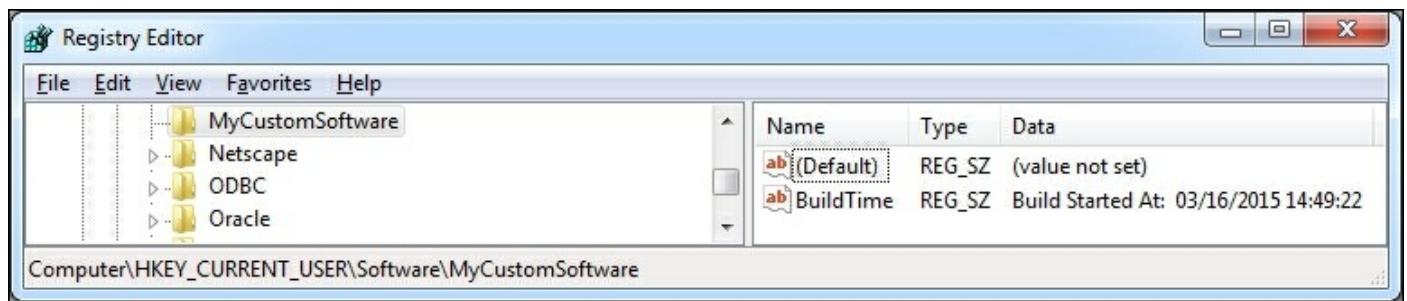
To create a named value in the registry, do the following action:

```
$regpath = "HKCU:\Software\MyCustomSoftware\"  
$regname = "BuildTime"  
$regvalue = "Build Started At: " + " " + (get-date)  
New-itemproperty -path $regpath -name $regname -PropertyType  
String -value $regvalue  
$verifyValue = Get-itemproperty -path $regpath -name $regname  
Write-Host "The $regName named value is set to: "  
$verifyValue.$regname
```

The output of this is shown in the following screenshot:

```
PS C:\> $regpath = "HKCU:\Software\MyCustomSoftware\"  
PS C:\> $regname = "BuildTime"  
PS C:\> $regvalue = "Build Started At: " + " " + (get-date)  
PS C:\> New-ItemProperty -path $regpath -name $regname -PropertyType String -value $regvalue  
  
BuildTime      : Build Started At: 03/16/2015 14:49:22  
PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\MyCustomSoftware\  
PSParentPath   : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software  
PSChildName    : MyCustomSoftware  
PSDrive        : HKCU  
PSProvider     : Microsoft.PowerShell.Core\Registry  
  
PS C:\> $verifyValue = Get-ItemProperty -path $regpath -name $regname  
PS C:\> Write-Host "The $regName named value is set to: " $verifyValue.$regname  
The BuildTime named value is set to: Build Started At: 03/16/2015 14:49:22
```

After executing the script, the registry will look like the following screenshot:



This script displays how you can create a registry named value in a specific location. It also displays how you can retrieve a value and display it in the console. You first start by defining several variables. The first variable `$regpath` defines where you want to create the new named value which is in the `HKCU:\Software\MyCustomSoftware\` registry key. The second variable `$regname` defines what you want the new named value to be named, which is `BuildTime`. The third variable defines what you want the value of the named value to be, which is `Build Started At:` with the current date and time. The next step in the script is to create the new value. You first call the `New-ItemProperty` cmdlet with the `-path` argument and specify the `$regpath` variable. You then use the `-name` argument and specify `$regname`. This is followed by specifying the

-PropertyType argument and by specifying the string PropertyType. Finally, you specify the -value argument and use the \$regvalue variable to fill the named value with data.

Proceeding forward in the script, you verify that the named value has proper data by leveraging the get-itemproperty cmdlet. You first define the \$verifyvalue variable that captures the data from the cmdlet. You then issue get-itemproperty with the -path argument of \$regpath and the -name argument of \$regname. You then write to the console that the \$regname named value is set to \$verifyvalue.\$regname. When you are done with script execution, you should have a new registry named value of BuildTime in the HKEY_CURRENT_USER:\Software\MyCustomSoftware\ key with a value similar to Build Started At: 03/16/2015 14:49:22.

Verifying files, folders, and registry items

When you are creating and modifying objects, it's important to make sure that the file, folder, and registry items don't exist prior to creating and modifying them. The `test-path` cmdlet allows you to test to see if a file, folder, or registry item exists prior to working with it. The proper syntax for this is first calling `test-path` and then specifying a file, folder, or registry location. The result of the `test-path` command is `True` if the object exists or `False` if the object doesn't exist.

To verify if files, folders, and registry entries exist, do the following action:

```
$testfolder = test-path "c:\Program  
Files\MyCustomSoftware\Logs"  
#Update The Following Line with the Date/Timestamp of your file  
$testfile = test-path "c:\Program  
Files\MyCustomSoftware\Logs\201503163824LogFile.txt"  
$testreg = test-path "HKCU:\Software\MyCustomSoftware\"  
If ($testfolder) { write-host "Folder Found!" }  
If ($testfile) { write-host "File Found!" }  
If ($testreg) { write-host "Registry Key Found!" }
```

The output is shown in the following screenshot:

```
PS C:\> $testfolder = test-path "c:\Program Files\MyCustomSoftware\Logs"  
PS C:\> #Update The Following Line with the Date/Timestamp of your file  
PS C:\> $testfile = test-path "c:\Program Files\MyCustomSoftware\Logs\201503163824LogFile.txt"  
PS C:\> $testreg = test-path "HKCU:\Software\MyCustomSoftware\"  
PS C:\> If ($testfolder) { write-host "Folder Found!" }  
Folder Found!  
PS C:\> If ($testfile) { write-host "File Found!" }  
File Found!  
PS C:\> If ($testreg) { write-host "Registry Key Found!" }  
Registry Key Found!
```

This example displays how to verify if a file, folder, and registry item exists. You first start by declaring a variable to catch the output from the `test-path` cmdlet. You then specify `test-path`, followed by a file,

folder, or registry item whose existence you want to verify.

In this example, you start by using the `test-path` cmdlet to verify if the `Logs` folder is located in the `c:\Program Files\MyCustomSoftware\` directory. You then store the result in the `$testfolder` variable. You then use the `test-path` cmdlet to check if the file located at `c:\Program Files\MyCustomSoftware\Logs\201503163824LogFile.txt` exists. You then store the result in the `$testfile` variable. Finally, you use the `test-path` cmdlet to see if the registry key of

`HKCU:\Software\MyCustomSoftware\` exists. You then store the result in the `$testreg` variable. To evaluate the variables, you create `IF` statements to check whether the variables are `True` and write to the console if the items are found. After executing the script, the console will output the messages `Folder Found!`, `File Found!`, and `Registry Key Found!`.

Copying and moving files and folders

When you are working in the operating system, there may be instances where you need to copy or move files and folders around on the operating system. PowerShell provides two cmdlets to copy and move files. The `copy-item` cmdlet allows you to copy a file or a folder from one location to another. The proper syntax of this cmdlet is calling `copy-item`, followed by `-path` argument for the source you want to copy and the `-destination` argument for the destination of the file or folder. The `copy-item` cmdlet also has the `-force` argument to write over a read-only or hidden file. There are instances when read-only files cannot be overwritten, such as a lack of user permissions, which will require additional code to change the file attributes before copying over files or folders. The `copy-item` cmdlet also has a `-recurse` argument, which allows you to recursively copy the files in a folder and its subdirectories.

Tip

A common trick to use with the `copy-item` cmdlet is to rename during the copy operation. To do this, you change the destination to the desired name you want the file or folder to be. After executing the command, the file or folder will have a new name in its destination. This reduces the number of steps required to copy and rename a file or folder.

The `move-item` cmdlet allows you to move files from one location to another. The `move-item` cmdlet has the same syntax as the `copy-item` cmdlet. The proper syntax of this cmdlet is calling `move-item`, followed by the `-path` argument for the source you want to move and the `-destination` argument for the destination of the file or folder. The `move-item` cmdlet also has the `-force` overload to write over a read-only or hidden file. There are also instances when read-only files cannot be overwritten, such as a lack of user permissions, which will require additional code to change the file attributes before moving files or folders. The `move-item` cmdlet does not, however, have a `-recurse`

argument. Also, it's important to remember that the `move-item` cmdlet requires the destination to be created prior to the move. If the destination folder is not available, it will throw an exception. It's recommended to use the `test-path` cmdlet in conjunction with the `move-item` cmdlet to verify that the destination exists prior to the move operation.

Tip

PowerShell has the same file and folder limitations as the core operating system it is being run on. This means that file paths that are longer than 256 characters in length will receive an error message during the copy process. For paths that are over 256 characters in length, you need to leverage `robocopy.exe` or a similar file copy program to copy or move files.

All `move-item` operations are recursive by default. You do not have to specify the `-recurse` argument to recursively move files. To copy files recursively, you need to specify the `-recurse` argument.

To copy and move files and folders, do the following action:

```
New-item -path "c:\Program Files\MyCustomSoftware\AppTesting" -  
ItemType Directory | Out-null  
New-item -path "c:\Program  
Files\MyCustomSoftware\AppTesting\Help" -ItemType Directory |  
Out-null  
New-item -path "c:\Program Files\MyCustomSoftware\AppTesting\"  
-name AppTest.txt -ItemType File | out-null  
New-item -path "c:\Program  
Files\MyCustomSoftware\AppTesting\Help\" -name  
HelpInformation.txt -ItemType File | out-null  
New-item -path "c:\Program Files\MyCustomSoftware\" -name  
ConfigFile.txt -ItemType File | out-null  
move-item -path "c:\Program Files\MyCustomSoftware\AppTesting"  
-destination "c:\Program Files\MyCustomSoftware\Archive" -force  
copy-item -path "c:\Program  
Files\MyCustomSoftware\ConfigFile.txt" "c:\Program  
Files\MyCustomSoftware\Archive\Archived_ConfigFile.txt" -force
```

The output of this is shown in the following screenshot:

```
PS C:\> New-item -path "c:\Program Files\MyCustomSoftware\AppTesting" -ItemType Directory | Out-null
PS C:\> New-item -path "c:\Program Files\MyCustomSoftware\AppTesting\Help" -ItemType Directory | Out-null
PS C:\> New-item -path "c:\Program Files\MyCustomSoftware\AppTesting\" -name AppTest.txt -ItemType File | Out-null
PS C:\> New-item -path "c:\Program Files\MyCustomSoftware\AppTesting\Help\" -name HelpInformation.txt -ItemType File | Out-null
PS C:\> New-item -path "c:\Program Files\MyCustomSoftware\" -name ConfigFile.txt -ItemType File | Out-null
PS C:\> copy-item -path "c:\Program Files\MyCustomSoftware\ConfigFile.txt" "c:\Program Files\MyCustomSoftware\Archive\Archived_ConfigFile.txt" -force
PS C:\> move-item -path "c:\Program Files\MyCustomSoftware\AppTesting" -destination "c:\Program Files\MyCustomSoftware\Archive" -force
```

This example displays how to properly use the `copy-item` and `move-item` cmdlets. You first start by using the `new-item` cmdlet with the `-path` argument set to `c:\Program Files\MyCustomSoftware\AppTesting` and the `-ItemType` argument set to `Directory`. You then pipe the command to `out-null` to suppress the default output. This creates the `AppTesting` sub directory in the `c:\Program Files\MyCustomSoftware\` directory. You then create a second folder using the `new-item` cmdlet with the `-path` argument set to `c:\Program Files\MyCustomSoftware\Help` and the `-ItemType` argument set to `Directory`. You then pipe the command to `out-null`. This creates the `Help` sub directory in the `c:\Program Files\MyCustomSoftware\` directory.

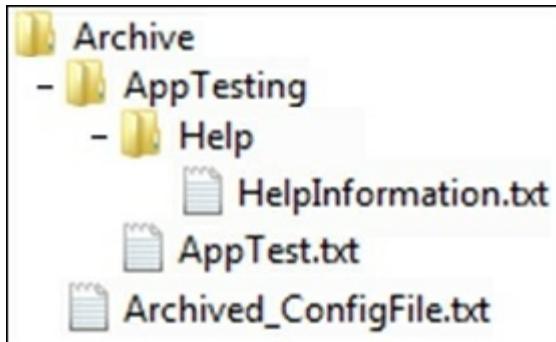
After creating the directories, you create a new file using the `new-item` cmdlet with the path of `c:\Program Files\MyCustomSoftware\AppTesting\`, the `-name` argument set to `AppTest.txt`, the `-ItemType` argument set to `File`; you then pipe it to `out-null`. You create a second file by using the `new-item` cmdlet with the path of `c:\Program Files\MyCustomSoftware\AppTesting\Help`, the `-name` argument set to `HelpInformation.txt` and the `-ItemType` argument set to `File`, and then piping it to `out-null`. Finally, you create a third file using the `new-item` cmdlet with the path of `c:\Program Files\MyCustomSoftware\`, the `-name` argument set to `ConfigFile.txt` and the `-ItemType` argument set to `File`, and then pipe it to `out-null`.

After creating the files, you are ready to start copying and moving files.

You first move the `AppTesting` directory to the `Archive` directory by using the `move-item` cmdlet and then specifying the `-path` argument with the value of `c:\Program Files\MyCustomSoftware\AppTesting` as the

source, the `-destination` argument with the value of `c:\Program Files\MyCustomSoftware\Archive` as the destination, and the `-force` argument to force the move if the directory is hidden. You then copy a configuration file by using the `copy-item` cmdlet, using the `-path` argument with `c:\Program Files\MyCustomSoftware\ConfigFile.txt` as the source, and then specifying the `-destination` argument with `c:\Program Files\MyCustomSoftware\Archive\Archived_ConfigFile.txt` as the new destination with a new filename; you then leverage the `-force` argument to force the copy if the file is hidden.

This follow screenshot displays the file folder hierarchy after executing this script:



After executing this script, the file folder hierarchy should be as displayed in the preceding screenshot. This also displays that when you move the `AppTesting` directory to the `Archive` folder, it automatically performs the move recursively, keeping the file and folder structure intact.

Renaming files, folders, registry keys, and named values

When you are working with PowerShell scripts, you may have instances where you need to rename files, folders, and registry keys. The `rename-item` cmdlet can be used to perform renaming operations on a system. The syntax for this cmdlet is `rename-item` and specifying the `-path` argument with path to the original object, and then you call the `-newname` argument with a full path to what you want the item to be renamed to. The `rename-item` has a `-force` argument to force the rename in instances where the file or folder is hidden or restricted by UAC or to avoid prompting for the rename action.

To copy and rename files and folders, do the following action:

```
New-item -path "c:\Program  
Files\MyCustomSoftware\OldConfigFiles\" -ItemType Directory |  
out-null  
Rename-item -path "c:\Program  
Files\MyCustomSoftware\OldConfigFiles" -newname "c:\Program  
Files\MyCustomSoftware\ConfigArchive" -force  
copy-item -path "c:\Program  
Files\MyCustomSoftware\ConfigFile.txt" "c:\Program  
Files\MyCustomSoftware\ConfigArchive\ConfigFile.txt" -force  
Rename-item -path "c:\Program  
Files\MyCustomSoftware\ConfigArchive\ConfigFile.txt" -newname  
"c:\Program  
Files\MyCustomSoftware\ConfigArchive\Old_ConfigFile.txt" -force
```

The output of this is shown in the following screenshot:

```
PS C:\> New-item -path "c:\Program Files\MyCustomSoftware\OldConfigFiles\" -ItemType Directory | out-null  
PS C:\> Rename-item -path "c:\Program Files\MyCustomSoftware\OldConfigFiles" -newname "c:\Program Files\MyCustomSoftware  
\ConfigArchive" -force  
PS C:\> copy-item -path "c:\Program Files\MyCustomSoftware\ConfigFile.txt" "c:\Program Files\MyCustomSoftware\ConfigArch  
ive\ConfigFile.txt" -force  
PS C:\> Rename-item -path "c:\Program Files\MyCustomSoftware\ConfigArchive\ConfigFile.txt" -newname "c:\Program Files\My  
CustomSoftware\ConfigArchive\Old_ConfigFile.txt" -force
```

In this example, you create a script that creates a new folder and a new

file, and then renames the file and the folder. To start, you leverage the `new-item` cmdlet which creates a new folder in `c:\Program Files\MyCustomSoftware\` named `OldConfigFiles`. You then pipe that command to `Out-Null`, which silences the standard console output of the folder creation. You proceed to rename the folder `c:\Program Files\MyCustomSoftware\OldConfigFiles` with the `rename-item` cmdlet using the `-newname` argument to `c:\Program Files\MyCustomSoftware\ConfigArchive`. You follow the command with the `-force` argument to force the renaming of the folder.

You leverage the `copy-item` cmdlet to copy the `ConfigFile.txt` into the `ConfigArchive\` directory. You first start by specifying the `copy-item` cmdlet with the `-path` argument set to `c:\Program Files\MyCustomSoftware\ConfigFile.txt` and the destination set to `c:\Program Files\MyCustomSoftware\ConfigArchive\ConfigFile.txt`. You include the `-Force` argument to force the copy.

After moving the file, leverage the `rename-item` cmdlet with the `-path` argument to rename `c:\Program Files\MyCustomSoftware\ConfigArchive\ConfigFile.txt` using the `-newname` argument to `c:\Program Files\MyCustomSoftware\ConfigArchive\Old_ConfigFile.txt`. You follow this command with the `-force` argument to force the renaming of the file. At the end of this script, you will have successfully renamed a folder, moved a file into that renamed folder, and renamed a file in the newly created folder.

In the instance that you want to rename a registry key, do the following action:

```
New-item -path "HKCU:\Software\MyCustomSoftware\" -name CInfo -force | out-null  
Rename-item -path "HKCU:\Software\MyCustomSoftware\CInfo" -newname ConnectionInformation -force
```

The output of this is shown in the following screenshot:

```
PS C:\> New-item -path "HKCU:\Software\MyCustomSoftware\" -name CInfo -force | out-null  
PS C:\> Rename-item -path "HKCU:\Software\MyCustomSoftware\CInfo" -newname ConnectionInformation -force
```

After renaming the subkey, the registry will look like the following screenshot:



This example displays how to create a new subkey and rename it. You first start by using the `new-item` cmdlet to create a new sub key with the `-path` argument of the `HKCU:\Software\MyCustomSoftware\` key and the `-name` argument set to `CInfo`. You then pipe that line to `out-null` in order to suppress the standard output from the script. You proceed to execute the `rename-item` cmdlet with the `-path` argument set to `HKCU:\Software\MyCustomSoftware\CInfo` and the `-newname` argument set to `ConnectionInformation`. You then use the `-force` argument to force the renaming in instances when the subkey is restricted by UAC. After executing this command, you will see that the `CInfo` subkey located in `HKCU:\Software\MyCustomSoftware\` is now renamed to `ConnectionInformation`.

When you want to update named values in the registry, you will not be able to use the `rename-item` cmdlet. This is due to the named values being properties of the keys themselves. Instead, PowerShell provides the `rename-itemproperty` cmdlet to rename the named values in the key. The proper syntax for this cmdlet is calling `rename-itemproperty` by using the `-path` argument, followed by the path to the key that contains the named value. You then issue the `-name` argument to specify the named value you want to rename. Finally, you specify the `-newname` argument and the name you want the named value to be renamed to.

To rename the registry named value, do the following action:

```
$regpath =  
"HKCU:\Software\MyCustomSoftware\ConnectionInformation"
```

```

$regname = "DBServer"
$regvalue = "mySQLserver.mydomain.local"
New-itemproperty -path $regpath -name $regname -PropertyType
String -value $regvalue | Out-null
Rename-itemproperty -path $regpath -name DBServer -newname
DatabaseServer

```

The output of this is shown in the following screenshot:

```

PS C:\> $regpath = "HKCU:\Software\MyCustomSoftware\ConnectionInformation"
PS C:\> $regname = "DBServer"
PS C:\> $regvalue = "mySQLserver.mydomain.local"
PS C:\> New-itemproperty -path $regpath -name $regname -PropertyType String -value $regvalue | Out-null
PS C:\> Rename-itemproperty -path $regpath -name DBServer -newname DatabaseServer

```

After updating the named value, the registry will reflect this change, and so should look like the following screenshot:

 DatabaseServer	REG_SZ	mySQLserver.mydomain.local
--	--------	----------------------------

The preceding script displays how to create a new named value and rename it to a different named value. You first start by defining the variables to be used with the `new-itemproperty` cmdlet. You define the location of the registry subkey in the `$regpath` variable and set it to `HKCU:\Software\MyCustomSoftware\ConnectionInformation`. You then specify the named value name of `DBServer` and store it in the `$regname` variable. Finally, you define the `$regvalue` variable and store the value of `mySQLserver.mydomain.local`.

To create the new named value, leverage `new-itemproperty`, specify the `-path` argument with the `$regpath` variable, use the `-name` argument with the `$regname` variable, and use the `-value` argument with the `$regvalue` variable. You then pipe this command to `out-null` in order to suppress the default output of the command. This command will create the new named value of `DBServer` with the value of `mySQLserver.mydomain.local` in the `HKCU:\Software\MyCustomSoftware\ConnectionInformation` subkey.

The last step in the script is renaming the DBServer named value to DatabaseServer. You first start by calling the `rename-itemproperty` cmdlet and then using the `-path` argument and specifying the `$regpath` variable which contains the

`HKCU:\Software\MyCustomSoftware\ConnectionInformation` subkey; you then proceed by calling the `-name` argument and specifying `DBServer` and finally calling the `-newname` argument with the new value of `DatabaseServer`. After executing this command, you will see that the `HKCU:\Software\MyCustomSoftware\ConnectionInformation` key has a new named value of `DatabaseServer` containing the same value of `mySQLserver.mydomain.local`.

Deleting files, folders, registry keys, and named values

When you are creating scripts, there are instances when you need to delete items from a computer. PowerShell has the `remove-item` cmdlet that enables the removal of objects from a computer. The syntax of this cmdlet starts by calling the `remove-item` cmdlet and proceeds with specifying the `-path` argument with a file, folder, or registry key to delete.

The `remove-item` cmdlet has several useful arguments that can be leveraged. The `-force` argument is available to delete files, folders, and registry keys that are read-only, hidden, or restricted by UAC. The `-recurse` argument is available to enable recursive deletion of files, folders, and registry keys on a system. The `-include` argument enables you to delete specific files, folders, and registry keys. The `-include` argument allows you to use the wildcard character of an asterisk (*) to search for specific values in an object name or a specific object type. The `-exclude` argument will exclude specific files, folders, and registry keys on a system. It also accepts the wildcard character of an asterisk (*) to search for specific values in an object name or a specific object type.

The named values in the registry are properties of the key that they are contained in. As a result, you cannot use the `remove-item` cmdlet to remove them. Instead, PowerShell offers the `remove-itemproperty` cmdlet to enable the removal of the named values. The `remove-itemproperty` cmdlet has arguments similar to those of the `remove-item` cmdlet. It is important to note, however, that the `-filter`, `-include`, and `-exclude` arguments will not work with named values in the registry. They only work with item paths such as registry keys.

To set up the system for the deletion example, you need to process the following script:

```

# Create New Directory
new-item -path "c:\program files\MyCustomSoftware\Graphics\" -ItemType Directory | Out-null
# Create Files for This Example
new-item -path "c:\program files\MyCustomSoftware\Graphics\" -name FirstGraphic.bmp -ItemType File | Out-Null
new-item -path "c:\program files\MyCustomSoftware\Graphics\" -name FirstGraphic.png -ItemType File | Out-Null
new-item -path "c:\program files\MyCustomSoftware\Graphics\" -name SecondGraphic.bmp -ItemType File | Out-Null
new-item -path "c:\program files\MyCustomSoftware\Graphics\" -name SecondGraphic.png -ItemType File | Out-Null
new-item -path "c:\program files\MyCustomSoftware\Logs\" -name 2013010101LogFile.txt -ItemType File | Out-Null
new-item -path "c:\program files\MyCustomSoftware\Logs\" -name 2013020101LogFile.txt -ItemType File | Out-Null
new-item -path "c:\program files\MyCustomSoftware\Logs\" -name 2013030101LogFile.txt -ItemType File | Out-Null
# Create New Registry Keys and Named Values
New-item -path "HKCU:\Software\MyCustomSoftware\.AppSettings" | Out-null
New-item -path
"HKCU:\Software\MyCustomSoftware\ApplicationSettings" | Out-null
New-itemproperty -path
"HKCU:\Software\MyCustomSoftware\ApplicationSettings" -name AlwaysOn -PropertyType String -value True | Out-null
New-itemproperty -path
"HKCU:\Software\MyCustomSoftware\ApplicationSettings" -name AutoDeleteLogs -PropertyType String -value True | Out-null

```

The output of this is shown in the following screenshot:

```

PS C:\> # Create New Directory
PS C:\> new-item -path "c:\program files\MyCustomSoftware\Graphics\" -ItemType Directory | Out-null
PS C:\> # Create Files for This Example
PS C:\> new-item -path "c:\program files\MyCustomSoftware\Graphics\" -name FirstGraphic.bmp -ItemType File | Out-Null
PS C:\> new-item -path "c:\program files\MyCustomSoftware\Graphics\" -name FirstGraphic.png -ItemType File | Out-Null
PS C:\> new-item -path "c:\program files\MyCustomSoftware\Graphics\" -name SecondGraphic.bmp -ItemType File | Out-Null
PS C:\> new-item -path "c:\program files\MyCustomSoftware\Graphics\" -name SecondGraphic.png -ItemType File | Out-Null
PS C:\> new-item -path "c:\program files\MyCustomSoftware\Logs\" -name 2013010101LogFile.txt -ItemType File | Out-Null
PS C:\> new-item -path "c:\program files\MyCustomSoftware\Logs\" -name 2013020101LogFile.txt -ItemType File | Out-Null
PS C:\> new-item -path "c:\program files\MyCustomSoftware\Logs\" -name 2013030101LogFile.txt -ItemType File | Out-Null
PS C:\> # Create New Registry Keys and Named Values
PS C:\> New-item -path "HKCU:\Software\MyCustomSoftware\.AppSettings" | Out-null
PS C:\> New-item -path "HKCU:\Software\MyCustomSoftware\ApplicationSettings" | Out-null
PS C:\> New-itemproperty -path "HKCU:\Software\MyCustomSoftware\ApplicationSettings" -name AlwaysOn -PropertyType String -value True | Out-null
PS C:\> New-itemproperty -path "HKCU:\Software\MyCustomSoftware\ApplicationSettings" -name AutoDeleteLogs -PropertyType String -value True | Out-null

```

The preceding example is designed to set up the file structure for the following example. You first use the `new-item` cmdlet to create a new directory called `Graphics` in `c:\program files\MyCustomSoftware\`. You then use the `new-item` cmdlet to create new files named `FirstGraphic.bmp`, `FirstGraphic.png`, `SecondGraphic.bmp`, and `SecondGraphic.png` in the `c:\Program Files\MyCustomSoftware\Graphics\` directory. You then use the `new-item` cmdlet to create new log files in `c:\Program Files\MyCustomSoftware\Logs` named `2013010101LogFile.txt`, `2013020101LogFile.txt`, and `2013030101LogFile.txt`. After creating the files, you create two new registry keys located at `HKCU:\Software\MyCustomSoftware\AppSettings` and `HKCU:\Software\MyCustomSoftware\ApplicationSettings`. You then populate the `HKCU:\Software\MyCustomSoftware\ApplicationSettings` key with a named value of `AlwaysOn` set to `True` and a named value of `AutoDeleteLogs` set to `True`.

To remove files, folders, and registry items from a system, do the following action:

```
# Get Current year
$currentyear = get-date -f yyyy
# Build the Exclude String
$exclude = "*" + $currentyear + "*"
# Remove Items from System
Remove-item -path "c:\Program Files\MyCustomSoftware\Graphics\" -include *.bmp -force -recurse
Remove-item -path "c:\Program Files\MyCustomSoftware\Logs\" -exclude $exclude -force -recurse
Remove-itemproperty -path
"HKCU:\Software\MyCustomSoftware\ApplicationSettings" -Name
AutoDeleteLogs
Remove-item -path
"HKCU:\Software\MyCustomSoftware\ApplicationSettings"
```

The output of this is shown in the following screenshot:

```
PS C:\> # Get Current year
PS C:\> $currentyear = get-date -f yyyy
PS C:\> # Build the Exclude String
PS C:\> $exclude = "*" + $currentyear + "*"
PS C:\> # Remove Items from System
PS C:\> Remove-item -path "c:\Program Files\MyCustomSoftware\Graphics\" -include *.bmp -force -recurse
PS C:\> Remove-item -path "c:\Program Files\MyCustomSoftware\Logs\" -exclude $exclude -force -recurse
PS C:\> Remove-itemproperty -path "HKCU:\Software\MyCustomSoftware\ApplicationSettings" -Name AutoDeleteLogs
PS C:\> Remove-item -path "HKCU:\Software\MyCustomSoftware\ApplicationSettings"
```

This script displays how you can leverage PowerShell to clean up files and folders with the `remove-item` cmdlet and the `-exclude` and `-include` arguments. You first start by building the exclusion string for the `remove-item` cmdlet. You retrieve the current year by using the `get-date` cmdlet with the `-f` parameter set to `yyyy`. You save the output into the `$currentyear` variable. You then create a `$exclude` variable that appends asterisks on each end of the `$currentyear` variable, which contains the current date. This will allow the exclusion filter to find the year anywhere in the file or folder names.

The first command is that you use the `remove-item` cmdlet and call the `-path` argument with the path of `c:\Program Files\MyCustomSoftware\Graphics\`. You then specify the `-include` argument with the value of `*.bmp`. This tells the `remove-item` cmdlet to delete all files that end in `.bmp`. You then specify `-force` to force the deletion of the files and `-recurse` to search the entire `Graphics` directory to delete the files that meet the `*.bmp` inclusion criteria but leaves the other files you created with the `*.png` extension.

The second command leverages the `remove-item` cmdlet with the `-path` argument set to `c:\Program Files\MyCustomSoftware\Logs\`. You use the `-exclude` argument with the value of `$exclude` to exclude files that contain the current year. You then specify `-force` to force the deletion of the files and `-recurse` to search the entire `logs` directory to delete the files and folders that do not meet the exclusion criteria.

The third command leverages the `remove-itemproperty` cmdlet with the `-path` argument set to `HKCU:\Software\MyCustomSoftware\ApplicationSettings` and the `-name` argument set to `AutoDeleteLogs`. After execution, the

`AutoDeleteLogs` named path is deleted from the registry.

The last command leverages the `remove-item` cmdlet with the `-path` argument set to

`HKCU:\Software\MyCustomSoftware\ApplicationSettings`. After running this last command, the entire subkey of `ApplicationSettings` is removed from `HKCU:\Software\MyCustomSoftware\`.

After executing this script, you will see that the script deletes the `.BMP` files in the `C:\Program Files\MyCustomSoftware\Graphics` directory, but it leaves the `.PNG` files. You will also see that the script deletes all of the log files except the ones that had the current year contained in them. Last, you will see that the `ApplicationSettings` sub key that was created in the previous step is successfully deleted from

`HKCU:\Software\MyCustomSoftware\`.

Note

When you use the `remove-item` and `-recurse` parameters together, it is important to note that if `remove-item` cmdlet deletes all the files and folders in a directory, the `-recurse` parameter will also delete the empty folder and subfolders that contained those files. This is only true when there are no remaining files in the folders in a particular directory. This may create undesirable results on your system, and so you should use caution while performing this combination.

Summary

This chapter thoroughly explained the interaction of PowerShell with the files, folders, and registry objects. It began by displaying how to create a folder and a registry key by leveraging the `new-item` cmdlet. It also displayed the additional arguments that can be used with the `new-item` cmdlet to create a log file with the date time integrated in the filename. The chapter proceeded to display how to create and view a registry key property using the `get-itemproperty` and `new-itemproperty` cmdlets.

This chapter then moved to verification of files, folder, and registry items through the `test-path` cmdlet. By using this cmdlet, you can test to see if the object exists prior to interacting with it. You also learned how to interact with copying and moving files and folders by leveraging the `copy-item` and `move-item` cmdlets. You also learned how to rename files, folders, registry keys and registry properties with the use of the `rename-item` and `rename-itemproperty` cmdlets. This chapter ends with learning how to delete files, folders, and registry items by leveraging the `remove-item` and `remove-itemproperty` cmdlets. In the next chapter, you'll learn about file, folder, and registry attributes, access control lists, and properties. You'll learn how to fully modify these items with the use of PowerShell cmdlets.

Chapter 9. File, Folder, and Registry Attributes, ACLs, and Properties

In the previous chapter, you learned how to create, manage, and test the existence of files, folders, and registry items. You also learned how to rename these items and even copy and move the items to a new location. You ended the chapter by learning how to delete these items from a system.

This chapter extends what you learned in the previous chapter by providing an in-depth view into the attributes, properties, and **access control lists (ACL)** for files, folders, and registry items.

In this chapter, you will learn the following techniques:

- Retrieving attributes and properties for file, folder, and registry items
- Viewing file and folder extended attributes
- Setting mode and extended file and folder attributes
- Managing file, folder, and registry permissions
- Copying access control lists
- Adding and removing access control list rules

Note

The examples in this chapter require you to run PowerShell as administrator. You will not be able to successfully execute the examples if you don't run the PowerShell console as administrator.

This chapter uses script examples that build on the previous chapter. If you have not executed all the steps in the previous chapter, you can run the following script to set up the files, folders, and registry for this chapter:

```
# If the files, folders, and registry items don't exist, create
```

```

them.

if (!(test-path
"HKCU:\Software\MyCustomSoftware\ConnectionInformation")) {
New-item -path
"HKCU:\Software\MyCustomSoftware\ConnectionInformation" -force
| out-null }
if (!(test-path "HKCU:\Software\MyCustomSoftware\.AppSettings"))
{ New-item -path "HKCU:\Software\MyCustomSoftware\.AppSettings"
-force | out-null }
if (!(test-path "c:\Program Files\MyCustomSoftware\Graphics\")) {
New-item -path "c:\Program Files\MyCustomSoftware\" -name
Graphics -ItemType Directory | out-null }
if (!(test-path "c:\Program Files\MyCustomSoftware\Logs\")) {
New-item -path "c:\Program Files\MyCustomSoftware\" -name Logs
-ItemType Directory | out-null }
if (!(test-path "c:\Program
Files\MyCustomSoftware\Graphics\FirstGraphic.png")) { New-item
-path "c:\Program Files\MyCustomSoftware\Graphics\" -name
"FirstGraphic.png" -ItemType File | out-null }
if (!(test-path "c:\Program
Files\MyCustomSoftware\Graphics\SecondGraphic.png")) { New-item
-path "c:\Program Files\MyCustomSoftware\Graphics\" -name
"SecondGraphic.png" -ItemType File | out-null }

```

The output of this script is shown in the following screenshot:

```

PS C:\> # If the files, folders, and registry items don't exist, create them.
PS C:\> if (!(test-path "HKCU:\Software\MyCustomSoftware\ConnectionInformation")) { New-item -path "HKCU:\Software\MyCustomSoftware\ConnectionInformation" -force | out-null }
PS C:\> if (!(test-path "HKCU:\Software\MyCustomSoftware\.AppSettings")) { New-item -path "HKCU:\Software\MyCustomSoftware\AppSettings" -force | out-null }
PS C:\> if (!(test-path "c:\Program Files\MyCustomSoftware\Graphics\")){ New-item -path "c:\Program Files\MyCustomSoftware\" -name Graphics -ItemType Directory | out-null }
PS C:\> if (!(test-path "c:\Program Files\MyCustomSoftware\Logs\")){ New-item -path "c:\Program Files\MyCustomSoftware\" -name Logs -ItemType Directory | out-null }
PS C:\> if (!(test-path "c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png")) { New-item -path "c:\Program Files\MyCustomSoftware\Graphics\" -name "FirstGraphic.png" -ItemType File | out-null }
PS C:\> if (!(test-path "c:\Program Files\MyCustomSoftware\Graphics\SecondGraphic.png")) { New-item -path "c:\Program Files\MyCustomSoftware\Graphics\" -name "SecondGraphic.png" -ItemType File | out-null }

```

Retrieving attributes and properties

PowerShell provides the ability to view a structure of files, folders, and registry keys. This is performed by leveraging the `get-item` cmdlet. The proper syntax for using this cmdlet is calling `get-item`, followed by the `-`

`-path` trigger and the path to the file, folder, or registry location you want to look at. The result of the command will display the folders and files or the keys and subkeys for the registry.

When you use the `get-item` cmdlet, it evaluates only the actual object you are referencing. This means that if you reference `c:\windows` as your `-path` trigger, it will only return the properties of the folder itself. Likewise if you are attempting to view the properties of the registry key `HKLM:\Software\Microsoft\Windows\CurrentVersion\`, it will only display the named values and properties in that key and not the subkeys contained in that key.

To view the objects that are contained in a folder or registry key, you can use the `get-childitem` cmdlet. The proper syntax for this cmdlet starts by calling `get-childitem` and then proceeds with using the `-path` trigger and specifying a folder or registry key. After executing this command, you can interact with the child objects that are contained within that folder and registry key.

When you are interacting with files and folders using the `get-item` and `get-childitem` cmdlets, you will have the ability to see the file and folder mode attributes. These mode attributes provide the operating system-specific instructions on how to interact with the file and folder objects.

The available mode attributes for this include:

- `d----`: This directory attribute specifies that the object is a directory and can contain subdirectories and files.
- `-a---`: This archive attribute is used in backup scenarios to inform the back software if the file has changed since the last backup. When the archive attribute is present, the backup software will back up that file and clear the attribute.
- `--r-`: This read-only attribute specifies that the file or folder can only be read and the contents of that file or folder cannot be modified.
- `--h-`: This hidden attribute specifies that the file or folder objects

are hidden from view while exploring the filesystem. In order to see these items, you need to select **Show Hidden Files, Folders, or Drives** in the folder options, given that you've proper permissions to do so.

- **----s**: This system attribute is much like the hidden attribute where it will hide the file or folder from view while exploring the filesystem. This also indicates that the file or folder is integral to the functionality of the operating system and should not be changed or modified.

To view the properties of registry and folder items, you can perform the following action:

```
$regItem = get-item -path "HKCU:\Software\MyCustomSoftware\"  
$regItem  
$regChildItem = get-childitem -path  
"HKCU:\Software\MyCustomSoftware\"  
$regChildItem  
$dirItem = get-item -path "c:\Program  
Files\MyCustomSoftware\Graphics\"  
$dirItem  
$dirChildItem = get-childitem -path "c:\Program  
Files\MyCustomSoftware\Graphics\"  
$dirChildItem
```

The output of this script is shown in the following screenshot:

```

PS C:\> $regItem

Hive: HKEY_CURRENT_USER\Software

Name          Property
-----
MyCustomSoftware

PS C:\> $regChildItem = get-childitem -path "HKCU:\Software\MyCustomSoftware\"
PS C:\> $regChildItem

Hive: HKEY_CURRENT_USER\Software\MyCustomSoftware

Name          Property
-----
AppSettings
ConnectionInformation

PS C:\> $dirItem = get-item -path "c:\Program Files\MyCustomSoftware\Graphics\
PS C:\> $dirItem

Directory: C:\Program Files\MyCustomSoftware

Mode          LastWriteTime      Length Name
-----        -----           -----
d---  03/18/2015  9:47 AM           Graphics

PS C:\> $dirChildItem = get-childitem -path "c:\Program Files\MyCustomSoftware\Graphics\
PS C:\> $dirChildItem

Directory: C:\Program Files\MyCustomSoftware\Graphics

Mode          LastWriteTime      Length Name
-----        -----           -----
-a---  03/18/2015  9:47 AM           0 FirstGraphic.png
-a---  03/18/2015  9:47 AM           0 SecondGraphic.png

```

The preceding example displays how to properly view the attributes and properties of files, folders, and registry keys. It also displays how to view the attributes and properties of the child items of files, folders, and registry keys. You first start by declaring the variable of `$regitem` and then call the `get-item` cmdlet, with the `-path` trigger referencing the registry path of `HKCU:\Software\MyCustomSoftware\`. You then call `$regitem` which displays only the properties of the `MyCustomSoftware` subkey.

You then proceed to declare the `$regChildItem` variable with the `get-`

`childitem` cmdlet and supply the `-path` trigger referencing the registry path of `HKCU:\Software\MyCustomSoftware\`. You then call the `$regChildItem` variable, which will then display the child items of `MyCustomSoftware` including its subkeys and properties.

You continue the script by declaring the `$dirItem` variable, followed by the `get-item` cmdlet and the `-path` trigger pointing to `c:\Program Files\MyCustomSoftware\Graphics\`. You then call the `$diritem` variable which will display the attributes and properties of the `Graphics` directory. You will see that the `Mode` attribute is set to `d----`, which indicates that `Graphics` is a directory. You will also see the `LastWriteTime` attribute, which is the last time that an item was written or deleted in that directory.

Finally, you declare the `$dirChildItem` variable, followed by the `get-childitem` cmdlet. You reference the `-path` trigger and set the path to `c:\Program Files\MyCustomSoftware\Graphics\`. You then call the `$dirChildItem` variable, which will contain the child items of the `Graphics` directory. You will also see the properties of these child items with the `Mode` attribute of `-a---`, which indicates that these files have changed since the last backup of the system. You will also see the `LastWriteTime` attribute, which is the last time when those files were modified.

Viewing file and folder extended attributes

When you use the standard `get-item` and `get-childitem` cmdlets, you are able to see the default `mode` attributes for the files and folders that are available with FAT32 file systems. With the introduction of **New Technology File System (NTFS)**, however, Microsoft extended the file and folder attributes to a much larger set. This was done to support additional features and technologies surrounding NTFS such as encryption and compression.

The list of new attributes includes:

- `Compressed`: This attribute designates that the filesystem applied compression to the files or folders.
- `Encrypted`: This attribute designates that the filesystem applied encryption to the files or folders.
- `Normal`: This attribute, when assigned, clears the other attributes to make the files to have only the `NotContentIndexed` attribute and folders to have the `NotContentIndexed` and `Directory` attributes.
- `NotContentIndexed`: This attribute designates that the filesystem should include the file or folder as part of the routine indexing of the operating system so that searching can be expedited.
- `ReparsePoint`: A reparse point is a sector in the filesystem which designates user data for an application. A reparse point may also be a mounted volume designated by a folder.
- `SparseFile`: This attribute designates a large file that is made up of empty bit values. This could be a fixed size database or a virtual disk that is preprovisioned to be of a fixed size. While the file is reserving space contagiously, it may not be filled with data.
- `Temporary`: This attribute designates the files or folders to be temporary, and the operating system will parse the file or folders in memory while in use. This is designated for files and folders which have a very short lifetime on a system like the software installation source.

In order to expose all of the attributes, methods, and properties available to an object, you can leverage the `get-member` cmdlet. The `get-member` cmdlet is typically used in a piped scenario where you first reference an object using the `get-item` or `get-childitem` cmdlets with a file and then pipe the object to `get-member`. The `get-member` cmdlet will then display the Attributes, Properties, Methods, and other Extended Properties about that file.

To view the attributes, properties, methods, and extended properties of a file, do the following action:

```
get-item -path "c:\Program  
Files\MyCustomSoftware\Graphics\FirstGraphic.png" | get-member
```

The truncated output of this script is shown in the following screenshot:

TypeName: System.IO.FileInfo		
Name	MemberType	Definition
Mode	CodeProperty	System.String Mode{get=Mode;}
AppendText	Method	System.IO.StreamWriter AppendText()
CopyTo	Method	System.IO.FileInfo CopyTo(string destFileName), System.IO.FileInfo CopyTo(s...
Create	Method	System.IO.FileStream Create()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
CreateText	Method	System.IO.StreamWriter CreateText()
Decrypt	Method	void Decrypt()
Delete	Method	void Delete()
Encrypt	Method	void Encrypt()
Equals	Method	bool Equals(System.Object obj)
GetAccessControl	Method	System.Security.AccessControl.FileSecurity GetAccessControl(), System.Secur...

The preceding script shows how you display the Attributes, Properties, Methods, and other Extended Properties about a file. You first start by using the `get-item` cmdlet, with the `-path` trigger set to `c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png`. You then pipe that result to the `get-member` cmdlet. The result of this command is printing to the console a list of 48 different Attributes, Properties, Methods, and other Extended Properties. Some of the more notable attributes are the `Attributes` property, the `Mode` property, the `CodeProperty` property, the `LastAccessTime` property, the `LastWriteTime` property, the `Delete` method, the `Encrypt` method, the `Decrypt` method, and the `Open` method. Each of these items describes an action or information pertaining to the file located at `c:\Program`

Files\MyCustomSoftware\Graphics\FirstGraphic.png.

Setting the mode and extended file and folder attributes

When you want to change the mode and the extended attributes of a file or folder, you need to access the extended property named `Attributes`. The `Attributes` property allows you to read and write to itself. This means that if you want to replace all of the file or folder attributes, you can just declare the new `Attributes` and set them to the `Attributes` property. Likewise, if you want to maintain the existing `Attributes` but want to add new `Attributes`, you declare a variable and call the existing `Attributes` property; then you add the new `Attributes` using a comma separator. You then set that new variable to the `Attributes` property of a file or folder.

To view and add a new extended file attribute, do the following action:

```
# Get file attributes
$file = get-item -path "c:\Program
Files\MyCustomSoftware\Graphics\FirstGraphic.png"
$attributes = $file.attributes
$attributes
# Append ReadOnly attribute to existing attributes
$newattributes = "$attributes, ReadOnly"
# Write over existing attributes with new attributes
$file.attributes = $newattributes
$file.attributes
```

The output of this script is shown in the following screenshot:

```
PS C:\> # Get file attributes
PS C:\> $file = get-item -path "c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png"
PS C:\> $attributes = $file.attributes
PS C:\> $attributes
Archive
PS C:\> # Append ReadOnly attribute to existing attributes
PS C:\> $newattributes = "$attributes, ReadOnly"
PS C:\> # Write over existing attributes with new attributes
PS C:\> $file.attributes = $newattributes
PS C:\> $file.attributes
ReadOnly, Archive
```

After executing the preceding script, the FirstGraphic.png file will have properties as shown in the following screenshot:

Attributes: Read-only Hidden

This script displays how to retrieve the existing Attributes of a file, how to add the ReadOnly Attribute, and how to write the attribute back to the file. You first start by declaring the \$file variable. You then call the `get-item` cmdlet with the `-path` trigger set to `c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png`. This will set the `FirstGraphic.png` file object to the variable. You then declare the \$attributes variable and set that equal to the `Attributes` property by calling `$file.Attributes`. After that, you call the `$attributes` property to display the attributes to the console which reads `Archive, NotContentIndexed`.

You proceed to declare the `$newattributes` variable by setting it equal to the existing attributes of `$attributes` and by adding the comma separator and the Attribute of `ReadOnly`. You then set the `$file.attributes` property of `c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png` equal to `$newattributes`. You verify that the new attributes are set by calling the `$file.attributes` property again. After executing this script, you have successfully added the `ReadOnly` attribute to `c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png`.

When you want to remove attributes from files and folders, you typically want to preserve the existing attributes that have already been assigned to those items. This means that you cannot simply overwrite the file and folder attributes with the desired ones because each individual file and folder may have different attributes. The best way to handle this is to retrieve the attributes of a file or folder, place them into an array, loop through the attributes in the array, and rebuild a new array with only the attributes you want to use. You then set the attributes in that new array to the file or folder effectively removing the attributes

you no longer want on a file.

To parse file attributes and remove them, you can perform the following action:

```
# Get File Attributes
$file = get-item -path "c:\Program
Files\MyCustomSoftware\Graphics\FirstGraphic.png"
$attributes = $file.attributes
# Convert attributes to string
$attributes = $attributes.ToString()
# Split individual attributes into array
$attributes = $attributes.Split(",")
# Read through the individual attributes
Foreach ($attribute in $attributes) {
    # If read Only, skip
    if ($attribute -like "*ReadOnly*") {
        Write-Host "Skipping Attribute: $attribute"
    }
    # Else add attribute to attribute list
    else {
        $newattribute += "$attribute,"
        Write-Host "Attribute Added: $attribute"
    }
}
# Remove the trailing comma
$newattribute = $newattribute.TrimEnd(",")
# Write over existing attributes with new attributes
$file.attributes = $newattribute
Write-Host "New File attributes are: " $file.attributes
```

The output of this script is shown in the following screenshot:

```

PS C:\> # Get File Attributes
PS C:\> $file = get-item -path "c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png"
PS C:\> $attributes = $file.attributes
PS C:\> # Convert attributes to string
PS C:\> $attributes = $attributes.ToString()
PS C:\> # Split individual attributes into array
PS C:\> $attributes = $attributes.Split(",")
PS C:\> # Read through the individual attributes
PS C:\> Foreach ($attribute in $attributes) {
>> # If read Only, skip
>> if ($attribute -like "*ReadOnly*") {
>> write-host "Skipping Attribute: $attribute"
>>
>> # Else add attribute to attribute list
>> else {
>> $newattribute += "$attribute,"
>> Write-Host "Attribute Added: $attribute"
>>
>> }
>>
>> # Remove the trailing comma
>> $newattribute = $newattribute.TrimEnd(",")
>> # Write over existing attributes with new attributes
>> $file.attributes = $newattribute
>> Write-host "New File attributes are: " $file.attributes
>>
Skipping Attribute: ReadOnly
Attribute Added: Archive
New File attributes are: Archive

```

After executing the preceding script, the `FirstGraphic.png` file will have properties, as shown in the following screenshot:

Attributes: Read-only Hidden

The preceding script displays how to properly read the attributes of a file, store the attributes in a variable, identify the attribute to be removed, create a new variable with the attributes to be set, and set the attributes back on the file. To start, you first use the `get-item` cmdlet, with the `-path` trigger pointing to `c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png` and then save the object to `$file`. You then create the `$attributes` variable and populate the file's attributes by issuing `$file.attributes`. As the `Attributes` property is an object, you have to convert that property to text to parse the individual values. You then convert the object to a string by re-declaring the `$attributes` variable and using the `.ToString()` method on that variable. The result will have all of the attributes listed in a string format.

After converting the value to string, you have to use the `split()` method to place the individual Attributes as separate array items. You re-declare the `$attribute` variable and issue the `$attributes.split(",")` method to split the individual values into `$attributes` variables. From here you need to read through the individual attributes and create a `foreach` loop structure to evaluate each `$attribute` in the `$attributes` variable. You create an `IF` statement to evaluate if the current `$attribute` is -like `"*ReadOnly"`. If it is so, you write to the console Skipping Attribute: `$attribute`. If the value does not match `*ReadOnly`, then you add the existing `$attribute` being evaluated, followed by a comma to the `$newattribute` variable. You also write Attribute Added: `$attribute` to the console.

The `Attributes` property is very specific regarding the format of the `Attributes` property being passed into a file or a folder. As a result, you cannot have a trailing comma at the end of the `$newattribute` variable because the `Attribute` property will be looking for another `Attribute` value after that comma. To trim any trailing commas from the `$newattribute` variable, you execute the `$newattribute.trimend(",")` method, which removes the comma at the end of the variable. Finally, you set the `$file.attributes` property to the `$newattribute` variable and write New File Attributes are: `$file.attributes` to the console. After executing this script, you will see that `c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png` no longer has the `ReadOnly` attribute.

Tip

If a file or a folder has the `Hidden` attribute, you will not be able to interact with the file or folder using cmdlets. The workaround for this is to leverage the `-force` trigger in order to bypass the `Hidden` attribute. You will then be able to modify the hidden files and folders using the script described in this section.

Managing file, folder, and registry permissions

PowerShell provides the ability to manage file, folder, and registry permissions. To update security permissions, you have to interact with the access control list. The ACL contains access rules that permit or deny specific actions on a file, folder, or registry item. The most common security permissions that can be set on files and folders are shown here:

- **Full control:** This allows writing, reading, changing, and deleting objects.
- **Modify:** This allows writing, reading, and deleting objects. It does not permit taking ownership of objects.
- **Read and execute:** This allows reading, listing objects in a directory, and executing objects.
- **List folder contents:** This allows listing of objects in a directory and executing them.
- **Read:** This allows viewing and listing files and directories.
- **Write:** This allows the addition of new files and directories.

Note

Executing and reading files are two different actions. Reading is much like opening a PowerShell.ps1 file in ISE. You have the ability to view the code inside the file itself. Executing is like running a .ps1 file as a script within PowerShell. If you have read permissions but not executed them, Windows will block the execution of this script. It will, however, allow you to view the code in the script.

Copying access control lists

The process of copying the access control lists is for scenarios where you already have defined files, folders, and registry items, and you need to ensure that the additional objects have the same permissions as the other files, folders, and registry items. The method of copying access

control lists leverages the `get-acl` and `set-acl` cmdlets. The `get-acl` cmdlet is responsible for reading the access control list on a file, folder, or registry item. The proper syntax of this cmdlet is defining a variable, calling the `get-acl` cmdlet, and using the `-path` trigger to specify the location of an object. The variable will then contain the access control list of the object that you defined. When you want to change the ACL of a file or folder, you need to use the `set-acl` cmdlet with the variable that you defined to overlay the permissions on a different object. The proper syntax of this cmdlet includes calling the `set-acl` cmdlet, referencing the `-path` trigger with the path of the object who's ACL you want to change, followed by the `-aclobj` with the variable that contains the ACL. After executing these two cmdlets, you will have effectively copied permissions from one object to another.

To copy an access control list from one item to another, you can perform the following action:

```
# Get the existing ACL on the FirstGraphic.png file
$fileACL = get-acl -path "c:\Program
Files\MyCustomSoftware\Graphics\FirstGraphic.png"
# Set the ACL from FirstGraphic.png on SecondGraphic.png
Set-acl -path "c:\Program
Files\MyCustomSoftware\Graphics\SecondGraphic.png" -aclobj
$fileACL
# Get the existing ACL on the Logs directory
$dirACL = get-acl -path "c:\Program
Files\MyCustomSoftware\Logs"
# Set the ACL from the Logs directory on the Graphics directory
Set-acl -path "c:\Program Files\MyCustomSoftware\Graphics" -
aclobj $dirACL
# Get the existing ACL from the ConnectionInformation key
$regACL = get-acl -path
"HKCU:\Software\MyCustomSoftware\ConnectionInformation"
# Set the ACL from ConnectionInformation on the AppSettings key
Set-acl -path "HKCU:\Software\MyCustomSoftware\AppSettings" -
aclobj $regACL
```

The output of this is shown in the following screenshot:

```
PS C:\> # Get the existing ACL on the FirstGraphic.png file
PS C:\> $fileACL = get-acl -path "c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png"
PS C:\> # Set the ACL from FirstGraphic.png on SecondGraphic.png
PS C:\> Set-acl -path "c:\Program Files\MyCustomSoftware\Graphics\SecondGraphic.png" -aclobj $fileACL
PS C:\> # Get the existing ACL on the Logs directory
PS C:\> $dirACL = get-acl -path "c:\Program Files\MyCustomSoftware\Logs"
PS C:\> # Set the ACL from the Logs directory on the Graphics directory
PS C:\> Set-acl -path "c:\Program Files\MyCustomSoftware\Graphics" -aclobj $dirACL
PS C:\> # Get the existing ACL from the ConnectionInformation key
PS C:\> $regACL = get-acl -path "HKCU:\Software\MyCustomSoftware\ConnectionInformation"
PS C:\> # Set the ACL from ConnectionInformation on the AppSettings key
PS C:\> Set-acl -path "HKCU:\Software\MyCustomSoftware\AppSettings" -aclobj $regACL
```

This example displays how to properly copy an ACL from a file, folder, and registry key and apply these permissions to a different file, folder, and registry key. You start by declaring the `$fileACL` variable and then use the `get-acl` cmdlet and call the `-path` trigger pointing at the `c:\Program Files\MyCustomSoftware\Graphics\FirstGraphic.png` file. This copies the ACL from the file to the `$fileACL` variable. You then leverage the `set-acl` cmdlet and use the `-path` trigger pointing at the `c:\Program Files\MyCustomSoftware\Graphics\SecondGraphic.png` file; then you call `-aclobj` with the `$fileACL` variable. After executing these lines of code, the security permissions defined in the ACL of the `FirstGraphic.png` file have been copied over the permissions of the `SecondGraphic.png` file.

You then declare the `$dirACL` variable and use the `get-acl` cmdlet, followed by calling the `-path` trigger pointing at the `c:\Program Files\MyCustomSoftware\Logs` directory. This copies the ACL of that directory to the `$dirACL` variable. You then leverage the `set-acl` cmdlet and use the `-path` trigger pointing at the `c:\Program Files\MyCustomSoftware\Graphics\` directory, followed by calling `-aclobj` with the `$dirACL` variable. After executing this portion of the script, the security permissions defined in the ACL of the directory `Logs` are successfully copied over the permissions of the directory `Graphics`.

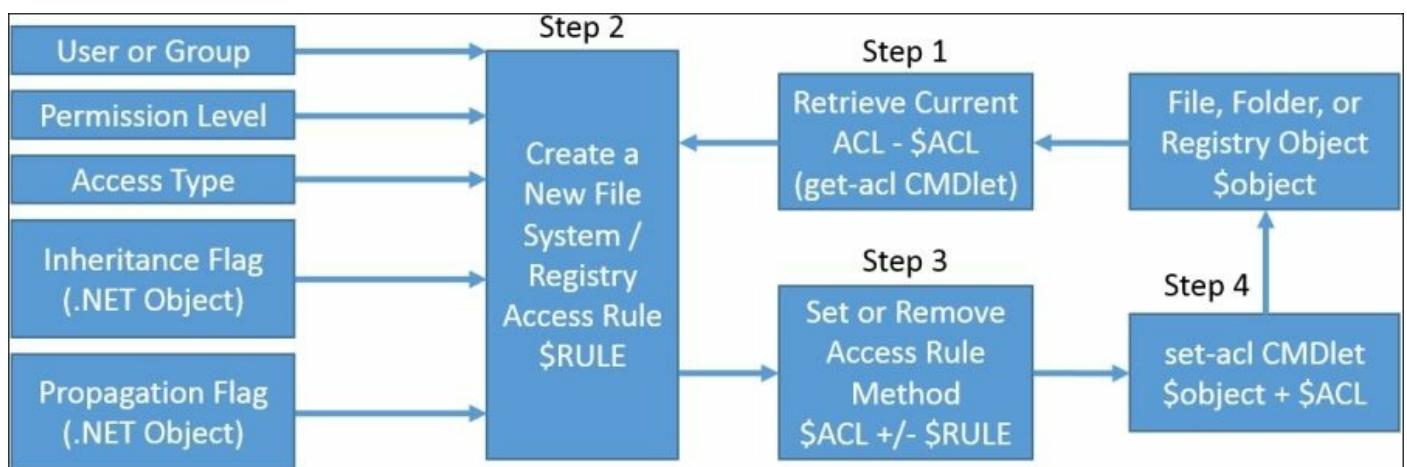
Finally, you set the permissions on the registry key. You first declare the `$regACL` variable, use the `get-acl` cmdlet, and call the `-path` trigger pointing at the `HKCU:\Software\MyCustomSoftware\ConnectionInformation` registry key. This copies the ACL of that key to the `$regACL` variable. You then

leverage the `set-acl` cmdlet and use the `-path` trigger pointing at the `HKCU:\Software\MyCustomSoftware\AppSettings` registry key, followed by calling `-aclobject` with the `$regACL` variable. After executing this portion of the script, the security permissions defined in the ACL of the `ConnectionInformation` registry key are successfully copied over the permissions of the `AppSettings` registry key.

Adding and removing ACL rules

PowerShell provides the ability to reference .NET classes in order to be able to perform advanced programming operations within scripts. While Microsoft has done a great job in expanding cmdlets to modify a large portion of the operating system, PowerShell doesn't provide the direct ability to add and remove individual access control list rules. This means that there isn't a direct method to add or remove a user or a group to a file, folder, or registry object.

The adding and removing ACL process flow is represented in the following diagram:



When you want to change an ACL, you are required to create a new access rule. An access rule defines permissions that are to be set or removed on a file, folder, or registry item. This is done by leveraging `system.security.accesscontrol.filesystemaccessrule` and the `system.security.accesscontrol.registryaccessrule` .NET class. This class is used in conjunction with the `get-acl` and `set-acl` cmdlets to

create and set access rules on particular objects.

To create a new access rule, you first have to obtain an ACL from a particular file, folder, or registry object. You will need to define a variable like \$ACL to contain the ACL of an object. You will then set the \$ACL variable equal to the `get-acl` cmdlet with the `-path` trigger pointing to a specific object.

The second step in the process is to create the access rule itself. Access rules require several components, which need to be declared for the rule to be properly set on an operating system. These requirements include the following objects:

- **User or group:** This is the user or group that you wanted to be added or removed from a file, folder, or registry object.
- **Inheritance flag:** This specifies that any new objects created in that directory have the permissions defined by the new access control rule. This will not apply to the existing objects in the directory. The inheritance flag must be set as a variable calling the `[system.security.accesscontrol.InheritanceFlags]` .NET class with the `ContainerInherit` and `ObjectInherit` arguments. This will ensure that the directories and subsequent objects will inherit the values(which are not required for registry access rules).
- **Propagation flag:** This specifies that the operating system should apply this permission to the subdirectories and subobjects recursively. This means that any file or folder below the object you are setting the new access control rule on will receive the new access rule. The propagation flag must be set as a variable calling the `[system.security.accesscontrol.PropagationFlags]` .NET class with the `none` argument. (This is not required for registry access rules.)
- **Permission level:** This specifies the permission level that you want to define for the user or group.
- **Access type:** This specifies whether you want the permission specified to be `Allow` or `Deny`.

The second step requires that you create a variable for the `$inherit`

requirement, which contains the

[`system.security.accesscontrol.InheritanceFlags`] "ContainerInheritObjectInherit" object. This will set the inheritance flag properly for use with the new access control rule. You also need to create a variable for the \$propagation requirement which contains the object

[`system.security.accesscontrol.PropagationFlags`] "None". This will set the propagation flag properly for use with the new access control rule.

To complete the second step, you have to create the access rule. To do this, you start by declaring a variable like \$rule to contain the new access rule. You then need to create a new object with the new-object cmdlet and reference the

`system.security.accesscontrol.filesystemaccessrule` or the `system.security.accesscontrol.registryaccessrule` .NET class. For file system permissions, you will provide the arguments required for the rule, which are these: (`User/Group`, `PermissionLevel`, `$inherit`, `$propagation`, `AccessType`). For registry permissions, you will provide the arguments required for the rule, which are these: (`User/Group`, `PermissionLevel`, `AccessType`). Upon successful entry of the required access rule items, the access rule will be stored in the \$rule variable.

The third step in the process is updating ACL variable with the new access rule. In the instances where you want to add or modify existing rules, you will use the `$ACL.setaccessrule($Rule)` method to modify the \$ACL variable to have the new Allow or Deny permissions. If you want to remove permissions, you will use the `$ACL.removeAccessRuleAll($Rule)` method to modify the \$ACL variable to not contain the items in the access rule.

The fourth and final step is leveraging the `set-acl` cmdlet with the `-path` trigger pointing to the file, folder, or registry item you want to modify. You also need to specify the `-aclobject` trigger with the \$ACL variable of which you want the new permissions to be. Upon execution of this portion of the script, the permissions will be added, modified, or removed from the file, folder, or registry item you specified.

To modify the access control list on a folder, you can perform the following action:

```
# Get the ACL from the Graphics directory
$ACL = Get-Acl "c:\program files\MyCustomSoftware\Graphics"
# Search the updated ACL for the Everyone group
$ACL.access | where { $_.IdentityReference -contains "Everyone" }
#
# Populate group variable for permissions
$group = "Everyone"
# Populate the permissions variable for setting permissions
$permission = "FullControl, Synchronize"
# Designate the inheritance information for permissions
$inherit =
[system.security.accesscontrol.InheritanceFlags] "ContainerInher
it, ObjectInherit"
# Designate the propagation information for permission
propagation
$propagation =
[system.security.accesscontrol.PropagationFlags] "None"
# Set to Allow Permissions
$accesstype = "Allow"
# Create the New Access Control list Rule
$Rule = New-Object
system.security.accesscontrol.filesystemaccessrule($group, $perm
ission, $inherit, $propagation, $accesstype)
# Merge new permissions with the existing ACL object
$ACL.SetAccessRule($RULE)
# Set the ACL on folder
Set-Acl -path "c:\program files\MyCustomSoftware\Graphics" -
aclobj $Acl
# Get Updated ACL on folder
$ACL = Get-Acl "c:\program files\MyCustomSoftware\Graphics"
# Search the updated ACL for the Everyone group
$ACL.access | where { $_.IdentityReference -contains "Everyone" }
```

The output of this script is shown in the following screenshot:

```

PS C:\> # Get the ACL from the Graphics directory
PS C:\> $ACL = Get-Acl "c:\program files\MyCustomSoftware\Graphics"
PS C:\> # Search the updated ACL for the Everyone group
PS C:\> $ACL.access | where { $_.IdentityReference -contains "Everyone" }
PS C:\> # Populate group variable for permissions
PS C:\> $group = "Everyone"
PS C:\> # Populate the permissions variable for setting permissions
PS C:\> $permission = "FullControl, Synchronize"
PS C:\> # Designate the inheritance information for permissions
PS C:\> $inherit = [system.security.accesscontrol.InheritanceFlags]"ContainerInherit, ObjectInherit"
PS C:\> # Designate the propagation information for permission propagation
PS C:\> $propagation = [system.security.accesscontrol.PropagationFlags]"None"
PS C:\> # Set to Allow Permissions
PS C:\> $accesstype = "Allow"
PS C:\> # Create the New Access Control list Rule
PS C:\> $Rule = New-Object system.security.accesscontrol.filesystemaccessrule($group,$permission,$inherit,$propagation,$accesstype)
PS C:\> # Merge new permissions with the existing ACL object
PS C:\> $ACL.SetAccessRule($RULE)
PS C:\> # Set the ACL on folder
PS C:\> Set-Acl -path "c:\program files\MyCustomSoftware\Graphics" -aclobject $ACL
PS C:\> # Get Updated ACL on folder
PS C:\> $ACL = Get-Acl "c:\program files\MyCustomSoftware\Graphics"
PS C:\> # Search the updated ACL for the Everyone group
PS C:\> $ACL.access | where { $_.IdentityReference -contains "Everyone" }

FileSystemRights : FullControl
AccessControlType : Allow
IdentityReference : Everyone
IsInherited : False
InheritanceFlags : ContainerInherit, ObjectInherit
PropagationFlags : None

```

This script displays how to obtain an existing ACL of a folder, how to create an access rule, how to apply the access rule to an existing ACL, and how to set the updated ACL back on the folder. This will successfully update permissions on the folder specified. You begin by obtaining the ACL of `c:\program files\MyCustomSoftware\Graphics` using the `get-acl` cmdlet and then proceed to storing the value in the `$ACL` variable. You then search the `$ACL.access` property to the `Graphics` folder and pipe that where there is an `IdentityReference` that matches the group named `Everyone`. Since the `Everyone` group is not currently assigned permissions on the folder, it will return nothing.

You then start building the prerequisites for the access rule by defining the group of `Everyone` to the `$group` variable and the permissions of `FullControl, Synchronize` in the `$permission` variable. You then create a new variable named `$inherit` and specify the .NET class

`[system.security.accesscontrol.InheritanceFlags]` with the `"ContainerInherit, ObjectInherit"` properties on the inheritance flag. This ensures that objects and other directories inherit the permissions as they are being created or moved to that location. You create a new variable named `$propagation` and specify the .NET class

`[system.security.accesscontrol.PropagationFlags]` with the `None`

properties on the propagation flag. This will ensure that the operating system will not propagate the permissions to the subdirectory or subobjects in that directory. You then create a variable of \$accesstype and set the access type property to Allow. Finally, you create the new access rule in the \$rule variable with the new-object cmdlet, referencing the .NET class

system.security.accesscontrol.filesystemaccessrule and the arguments of

(\$group, \$permission, \$inherit, \$propagation, \$accesstype).

After creating the access rule, you apply the access rule to the \$ACL variable using the setaccessrule method with the (\$Rule) argument. The \$Rule object will then apply to \$ACL, and the new permissions will be contained in the \$ACL variable. You then set the new ACL back on the c:\program files\MyCustomSoftware\Graphics folder with the set-acl cmdlet, the -path trigger to the folder location, and the -aclobject trigger pointing to the updated ACL in the \$ACL variable.

After applying the new ACL you then verify that the permissions were set properly by defining the \$ACL variable by using the get-acl cmdlet pointing at the c:\program files\MyCustomSoftware\Graphics folder. You then search the updated \$ACL.access property to the Graphics folder and pipe Where there is an IdentityReference that matches the group named Everyone. As the Everyone is now assigned permissions on the folder, the search will return with the Everyone group having FullControl, Synchronize permissions on the c:\program files\MyCustomSoftware\Graphics directory.

To modify the access control list of a registry key, you can perform the following action:

```
# Get the ACL from the ConnectionInformation registry key
$ACL = Get-Acl
"HKCU:\Software\MyCustomSoftware\ConnectionInformation"
# Search the updated ACL for the Everyone group
$ACL.access | where { $_.IdentityReference -contains "Everyone" }
#
# Populate group variable for permissions
$group = "Everyone"
```

```

# Populate the permissions variable for setting permissions
$permission = "FullControl"
# Set to Allow Permissions
$accesstype = "Allow"
# Create the New Access Control list Rule
$Rule = New-Object
system.security.accesscontrol.RegistryAccessrule($group,$permis
sion,$accesstype)
# Merge new permissions with the existing ACL object
$ACL.SetAccessRule($RULE)
# Set the ACL on registry key
Set-Acl -path
"HKCU:\Software\MyCustomSoftware\ConnectionInformation" --
aclobjetc $Acl
# Get Updated ACL on registry key
$ACL = Get-Acl
"HKCU:\Software\MyCustomSoftware\ConnectionInformation"
# Search the updated ACL for the Everyone group
$ACL.access | where { $_.IdentityReference -contains "Everyone"
}

```

The output of this script is shown in the following screenshot:

```

PS C:\> # Get the ACL from the ConnectionInformation registry key
PS C:\> $ACL = Get-Acl "HKCU:\Software\MyCustomSoftware\ConnectionInformation"
PS C:\> # Search the updated ACL for the Everyone group
PS C:\> $ACL.access | where { $_.IdentityReference -contains "Everyone" }
PS C:\> # Populate group variable for permissions
PS C:\> $group = "Everyone"
PS C:\> # Populate the permissions variable for setting permissions
PS C:\> $permission = "FullControl"
PS C:\>
PS C:\> # Set to Allow Permissions
PS C:\> $accesstype = "Allow"
PS C:\> # Create the New Access Control list Rule
PS C:\> $Rule = New-Object system.security.accesscontrol.RegistryAccessrule($group,$permission,$accesstype)
PS C:\> # Merge new permissions with the existing ACL object
PS C:\> $ACL.SetAccessRule($RULE)
PS C:\> # Set the ACL on registry key
PS C:\> Set-Acl -path "HKCU:\Software\MyCustomSoftware\ConnectionInformation" -aclobjetc $Acl
PS C:\> # Get Updated ACL on registry key
PS C:\> $ACL = Get-Acl "HKCU:\Software\MyCustomSoftware\ConnectionInformation"
PS C:\> # Search the updated ACL for the Everyone group
PS C:\> $ACL.access | where { $_.IdentityReference -contains "Everyone" }

RegistryRights      : FullControl
AccessControlType   : Allow
IdentityReference   : Everyone
IsInherited        : False
InheritanceFlags   : None
PropagationFlags   : None

```

This script displays how to obtain an existing ACL of a registry key, how to create an access rule and apply it to an existing ACL, and how to set the updated ACL back on the registry key. This will successfully update

permissions on the registry key specified. You begin by obtaining the ACL of `HKCU:\Software\MyCustomSoftware\ConnectionInformation` using the `get-acl` cmdlet, and then you store the value in the `$ACL` variable. You then search the `$ACL.access` property to the `ConnectionInformation` registry key and pipe that where there is `IdentityReference` that matches the group named `Everyone`. As the `Everyone` group is not currently assigned permissions on the registry key, it will return nothing.

You then start building the prerequisites for the access rule by defining the group of `Everyone` to the `$group` variable and the permissions of `FullControl` in the `$permission` variable. You then create a variable of `$accesstype` and set the access type property to `Allow`. Finally, you create the new access rule in the `$rule` variable with the `new-object` cmdlet, referencing the `system.security.accesscontrol.registryaccessrule` .NET class and the arguments of `($group, $permission, $accesstype)`.

After creating the access rule, you apply the access rule to the `$ACL` variable using the `setaccessrule` method with the `($Rule)` argument. The `$Rule` object will then apply to `$ACL`, and the new permissions will be contained in the `$ACL` variable. You then set the new ACL back on the `HKCU:\Software\MyCustomSoftware\ConnectionInformation` registry key with the `set-acl` cmdlet, the `-path` trigger to the registry key location, and the `-aclobject` trigger pointing to the updated ACL in the `$ACL` variable.

After applying the new ACL, you verify that the permissions were set properly by defining the `$ACL` variable using the `get-acl` cmdlet pointing at the `HKCU:\Software\MyCustomSoftware\ConnectionInformation` registry key. You then search the updated `$ACL.access` property to the `ConnectionInformation` registry key and pipe that where there is an `IdentityReference` that matches the group named `Everyone`. Since `Everyone` is now assigned permissions on the registry key, the search will return with the `Everyone` group having `FullControl` permissions on the `HKCU:\Software\MyCustomSoftware\ConnectionInformation` registry key.

Summary

This chapter thoroughly explained the interaction of PowerShell with the files, folders, and registry attributes and access control lists. It began by explaining how to use the `get-item` and `get-childitem` cmdlets to obtain the file, folder, and registry attributes. You also learned that these cmdlets are used to browse the subitems of the files, folders, and registry items. You then learned how to use the `get-item` cmdlet with the `get-member` cmdlet to list all of the available properties and methods available for a specific object type. You then proceeded to configure file attributes through the use of the built-in `Attribute` property of files. You learned how to remove attributes by converting the attribute's property to a string, splitting the values by the comma separator, creating a `foreach` loop to read through the individual attributes, and replacing the `Attributes` property of a file with the new attributes.

You also explored the `get-acl` and the `set-acl` cmdlets to copy permissions from one file, folder, or registry item to another. You learned that if you want to set permissions on a file or a folder, you need to follow four steps. First, you need to use the `get-acl` cmdlet to retrieve an existing access control list of the file, folder, or registry item you are trying to set permissions on. The second item is to create a filesystem or registry access rule. This rule is comprised of a user or a group, permission level, access type, and the inheritance and propagation flags for files and folders. The third step is to use the `setaccessrule()` or `removeaccessruleall()` methods to apply the new rule to the copied access control list. This will add, change, or remove the items specified in the access control list. You finally use the `set-acl` cmdlet to apply the new access control list to the file, folder, or registry item. You learned that in order to set permissions, you need to use the `System.Security.AccessControl.FileSystemAccessRule` or `System.Security.AccessControl.RegistryAccessRule` .NET classes to properly set permissions. Now, at the end of this chapter, you may be fully proficient in modifications of file, folders, and registry objects. In the next chapter, you'll explore **Windows Management**

Instrumentation (WMI). You'll learn how to leverage WMI and CIM cmdlets to view different classes and facilitate better management of systems.

Chapter 10. Windows Management Instrumentation

Windows Management Instrumentation (WMI) was created by Microsoft as a management engine for Windows-based operating systems. It provides the ability to view detailed information about a system's hardware and the operating system. WMI also provides the ability to perform actions on a computer, such as opening a program.

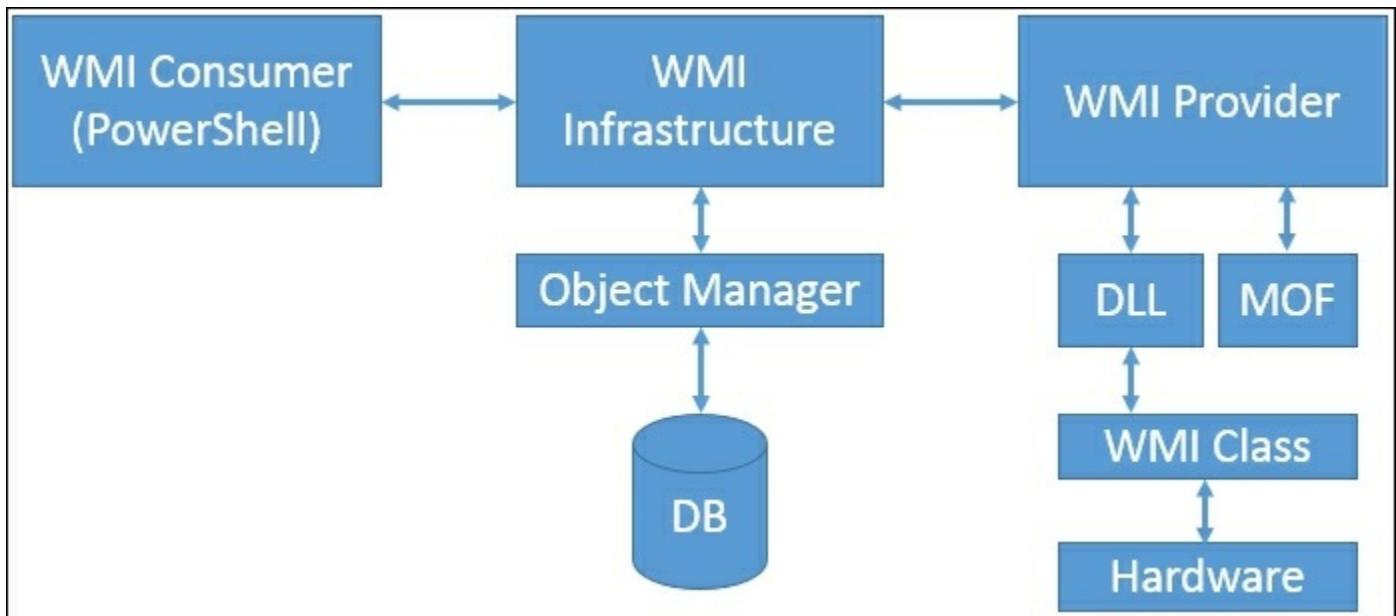
In this chapter, you will learn the following:

- WMI structure
- Using WMI objects
- Searching WMI classes
- Creating, modifying, and removing WMI object property instances
- Invoking WMI class methods

WMI structure

WMI is made up of three components. These three components include the **WMI consumers**, the **WMI infrastructure**, and the **WMI providers**. When you are using PowerShell, you will leverage all three of these components to interact with the hardware and operating system.

WMI structure is shown as follows:



WMI consumers are applications that can query and interact with the WMI. This may include PowerShell, .NET, C, C++, and other scripting and programming languages. Consumers communicate with the WMI infrastructure to obtain information about a system. WMI consumers do not, however, interact directly with the hardware or operating system through WMI.

The WMI infrastructure consists of an object manager and a WMI repository. The object manager keeps track of the used WMI instances on a system. The WMI repository is like a database, which keeps inventory of all the available WMI objects to interact with. These objects are imported into the **CIM Object Manager (CIMOM)**, via **Managed Object Format (MOF)** files. The MOF files provide the WMI infrastructure with a set of instructions on how the WMI consumers may interact with the hardware or the operating system. This can include fields that can be read, or actions that can be invoked on the hardware itself.

The WMI infrastructure, by default, places everything into a location, within the object manager, referred to as a **namespace**. The majority of the WMI objects that you will query will be created in the default namespace of `root\CIMv2`. Occasionally, applications may request the

WMI infrastructure to create a new namespace to place the information into. **System Center Configuration Manager (SCCM)**, for example, creates a new namespace of `root\CCM` when you install its client on a system. This namespace then maintains information collected about a system, software settings, and even methods that can be invoked by the SCCM.

The WMI provider is the actual driver that interacts with the operating system or hardware component. The WMI provider consists of a DLL and an MOF File. These provide information on what data is accessible by the WMI calls. They also provide information on the methods that can control the operating system or hardware components. The **drivers (DLL)** consist of classes which interact with the hardware and the operating system. When the WMI infrastructure interacts with the WMI provider, it is able to interact only with the items made available by the driver, and those specified in the MOF file.

Using WMI objects

When you leverage PowerShell to interact with WMI, you are interacting with WMI *namespaces* and WMI *classes*. WMI namespaces are collections of classes that represent management of a particular system. The Windows operating system, for example, uses the default `root\cimv2` namespace, and that collection contains over a thousand classes. Classes have multiple attributes such as properties and methods. Class properties represent information about that particular class, which is typically represented by a string or a numeric format. Methods are more like *actions*, which allow you to interact with that object, or other objects on a system.

The two primary cmdlets that allow you to retrieve a WMI object on a system are `get-wmiobject` and `get-ciminstance`. The `get-wmiobject` cmdlet is primarily used for querying a local system's WMI providers. You can leverage the `get-wmiobject` cmdlet by calling `get-wmiobject`, optionally defining a namespace with the `-namespace` parameter, and referencing a class name with the `-class` parameter. If you omit the `-namespace` parameter, the cmdlet assumes the default window's namespace of `root\cimv2`.

To properly leverage `get-wmiobject` cmdlet and retrieve a class, do the following:

```
get-wmiobject -namespace root\cimv2 -class win32_computerSystem
```

This command gives the output as shown in the following screenshot:

```
PS C:\> get-wmiobject -namespace root\cimv2 -class win32_computerSystem

Domain          : corp.cdw.com
Manufacturer    : LENOVO
Model           : 20BE0085US
Name            : LT-A0032132
PrimaryOwnerName : CDW
TotalPhysicalMemory : 16849174528
```

The preceding example displays how to retrieve a class using the `get-wmiobject` cmdlet. You first start by calling the `get-wmiobject` cmdlet. You then specify the `-namespace` parameter with the `root\cimv2` argument. Now you specify the `-class` parameter with the `win32_computersystem` class name argument. On executing the script, you will see the default properties from the `win32_computersystem` class.

The `get-ciminstance` cmdlet is very similar to the `get-wmiobject` cmdlet, though it has several differences. The `get-ciminstance` cmdlet has the ability to run on remote systems over a CIM session, whereas the `get-wmiobject` only permits local execution. By leveraging the `-cimsession` parameter, you can specify a CIM session to query a remote WMI instance. The `get-ciminstance` cmdlet also returns the same data in a different format. This is due to the `get-ciminstance` cmdlet being newer than the `get-wmiobject` cmdlet. This newer cmdlet follows the latest management specifications from the **Distributed Management Task Force (DMTF)**, and will display the results from commands according to the latest standards.

You can leverage the `get-ciminstance` cmdlet by calling `get-ciminstance`, optionally defining a namespace with the `-namespace` parameter, and referencing a class name with the `-class` parameter. If you omit the `-namespace` parameter, the cmdlet assumes the default window's namespace of `root\cimv2`.

To properly leverage the `get-ciminstance` cmdlet, and retrieve a class, do the following:

```
get-ciminstance -namespace root\cimv2 -class  
win32_computersystem
```

The output of this command is shown in the following screenshot:

PS C:\> get-ciminstance -namespace root\cimv2 -class win32_computersystem					
Name	PrimaryOwnerName	Domain	TotalPhysicalMemory	Model	Manufacturer
LT-A0032132	CDW	corp.cdw.com	16849174528	20BE0085US	LENOVO

The preceding example displays how to retrieve a class using the `get-ciminstance` cmdlet. You first start by calling the `get-ciminstance` cmdlet. Then you specify the `-namespace` parameter with the `root\cimv2` argument. Now you specify the `-class` parameter with the `win32_computerSystem` class name argument. On executing the script, you will see the default properties from the `win32_computerSystem` class.

Both, the `get-wmiobject` and the `get-ciminstance` cmdlets, list the default properties for classes when you leverage them. In most cases, additional properties and methods are contained in the class itself. If you want to dig deeper into the class, you can pipe the results to `get-member`, and it will display the full listing of all the attributes of that class.

To retrieve all the attributes of the `win32_computerSystem` class, do the following:

```
get-wmiobject -class win32_computerSystem | get-member
```

The following screenshot displays the output of this command:

TypeName: System.Management.ManagementObject#root\cimv2\Win32_ComputerSystem		
Name	MemberType	Definition
PSComputerName	AliasProperty	PSComputerName = __SERVER
JoinDomainOrWorkgroup	Method	System.Management.ManagementBaseObject JoinDomainOrWorkgroup(System.String...)
Rename	Method	System.Management.ManagementBaseObject Rename(System.String Name, System.S...
SetPowerState	Method	System.Management.ManagementBaseObject SetPowerState(System.UInt16 PowerSt...
UnjoinDomainOrWorkgroup	Method	System.Management.ManagementBaseObject UnjoinDomainOrWorkgroup(System.Stri...
AdminPasswordStatus	Property	uint16 AdminPasswordStatus {get;set;}
AutomaticManagedPagefile	Property	bool AutomaticManagedPagefile {get;set;}
AutomaticResetBootOption	Property	bool AutomaticResetBootOption {get;set;}
AutomaticResetCapability	Property	bool AutomaticResetCapability {get;set;}
BootOptionOnLimit	Property	uint16 BootOptionOnLimit {get;set;}

The preceding example displays how to retrieve all the class attributes using the `get-wmiobject` cmdlet and the `get-member` cmdlet. You first start by calling the `get-wmiobject` cmdlet. You then specify the `-namespace` parameter with the `root\cimv2` argument. Now you specify the `-class` parameter with the `win32_computerSystem` class name argument. Finally, you pipe the results to the `get-member` cmdlet, and you will see all the attributes of the `win32_computerSystem` class.

Searching for WMI classes

There are instances when you may want to search for different WMI classes on a system. The two primary cmdlets that enable you to search WMI are `get-wmiobject`, and `get-cimclass`. You can simply leverage the `get-wmiobject` cmdlet with the `-list` argument to list all the classes in a particular namespace. You can further narrow down the list by piping the command to the statement `where {$_ .Name -like "*Search*"}.` This will search the `Name` property of the classes that match a specific criterion.

An example of using the `get-wmiobject` cmdlet to find classes with a specific value would look like:

```
get-wmiobject -list | where{$_ .Name -like "*Time*"}
```

The output of this command is shown in the following screenshot:

```
PS C:\> get-wmiobject -list | where{$_ .Name -like "*Time*"}
```

NameSpace: ROOT\cimv2		
Name	Methods	Properties
__TimerNextFiring	{}	{NextEvent64BitTime, TimerId}
MSFT_NetConnectionTimeout	{}	{Milliseconds, SECURITY_DESCRIPTOR, Service, TIME_CREATED}
MSFT_NetTransactTimeout	{}	{Milliseconds, SECURITY_DESCRIPTOR, Service, TIME_CREATED}
MSFT_NetReadfileTimeout	{}	{Milliseconds, SECURITY_DESCRIPTOR, TIME_CREATED}
__TimerEvent	{}	{NumFirings, SECURITY_DESCRIPTOR, TIME_CREATED, TimerId}
__TimerInstruction	{}	{SkipIfPassed, TimerId}
AbsoluteTimerInstruction	{}	{EventDateTime, SkipIfPassed, TimerId}
IntervalTimerInstruction	{}	{IntervalBetweenEvents, SkipIfPassed, TimerId}
Win32_CurrentTime	{}	{Day, DayOfWeek, Hour, Milliseconds...}
Win32_LocalTime	{}	{Day, DayOfWeek, Hour, Milliseconds...}
Win32_UTCTime	{}	{Day, DayOfWeek, Hour, Milliseconds...}
Win32_TimeZone	{}	{Bias, Caption, DaylightBias, DaylightDay...}
Win32_SystemTimeZone	{}	{Element, Setting}

The preceding example displays how to properly leverage the `get-wmiobject` cmdlet to search for WMI classes. You first start by declaring the `get-wmiobject` cmdlet with the `-list` parameter. You then leverage where the pipeline property of `Name` is like the word `Time`. After executing this script, you will see all the class names that have the word `Time` in them.

Note

It is important to remember that when you are searching the full list of classes, it may take a few seconds to return results. This is due to the large number of classes available on the system, and having to evaluate the individual properties of these classes.

You may also choose the `get-cimclass` cmdlet to search for WMI classes. Like the other CIM-based cmdlets, the `get-cimclass` cmdlet supports sessions and allows you to query remote systems. You can simply leverage the `get-cimclass` cmdlet alone to return a full list of classes. You can further narrow down the list by piping the command to the statement `where {$_ .CimClassName -like "*Search*"}.` This will search the `CimClassName` property of the classes that match a specific criterion. You may also choose to use the `-cimsession` parameter and specify a CIM session to query the WMI on a remote system.

An example of using the `get-cimclass` cmdlet to find classes with a specific value would look like this:

```
get-cimclass | where{$_ .CimClassName -like "*Time*"}  
PS C:\> get-cimclass | where{$_ .CimClassName -like "*Time*"}  
NameSpace: ROOT/CIMV2  
CimClassName  
-----  
_TimerNextFiring  
MSFT_NetConnectionTimeout  
MSFT_NetTransactTimeout  
MSFT_NetReadfileTimeout  
_TimerEvent  
_TimerInstruction  
_AbsoluteTimerInstruction  
_IntervalTimerInstruction  
Win32_CurrentTime  
Win32_LocalTime  
Win32_UTCTime  
Win32_TimeZone  
Win32_SystemTimeZone  
CimClassMethods  
-----  
CimClassProperties  
-----  
[NextEvent64BitTime, TimerId]  
[SECURITY_DESCRIPTOR, TIME_CREATED, Milliseconds, Service]  
[SECURITY_DESCRIPTOR, TIME_CREATED, Milliseconds, Service]  
[SECURITY_DESCRIPTOR, TIME_CREATED, Milliseconds]  
[SECURITY_DESCRIPTOR, TIME_CREATED, NumFirings, TimerId]  
[SkipIfPassed, TimerId]  
[SkipIfPassed, TimerId, EventDateTime]  
[SkipIfPassed, TimerId, IntervalBetweenEvents]  
[Day, DayOfWeek, Hour, Milliseconds...]  
[Day, DayOfWeek, Hour, Milliseconds...]  
[Day, DayOfWeek, Hour, Milliseconds...]  
[Caption, Description, SettingID, Bias...]  
[Element, Setting]
```

The output of this command is shown in the following screenshot:

The preceding example displays how to properly leverage the `get-cimclass` cmdlet to search for WMI classes. You first start by declaring the `get-cimclass` cmdlet. You then leverage where the pipeline property of `CimClassName` is like the word `Time`. After executing this script, you will be left with all the class names that have the word `Time` in them.

You may also choose to leverage the `get-cimclass` cmdlets to query the class attributes. To start, you can use the `-class` parameter and declare a class name. Embracing the prior statement in parentheses, you can then leverage the dot notation to call either the `.CimClassProperties`, or `.CimClassMethods` attributes. This will list all the respective class properties or class methods.

To leverage the `get-cimclass` cmdlet to view the class properties, do the following:

```
$classProperties = (get-cimclass -class win32_Printer).CimClassProperties  
$classProperties.count
```

This set of commands gives the output as shown in the following screenshot:

```
PS C:\> $classProperties = (get-cimclass -class win32_Printer).CimClassProperties  
PS C:\> $classProperties.count  
86
```

This example displays the proper syntax for retrieving the `CimClassProperties` property of `win32_printer` class and counting the number of properties available for use. You first start by declaring the `$classProperties` variable and setting it to a parenthetically enclosed `get-cimclass` properties with the `-class` parameter value set to `win32_printer`. You then leverage the dot notation to view the `CimClassProperties` of that item. Finally, you retrieve the property `count` of that class by leveraging the `.count` property on the `$classProperties` variable. On executing this script, you will have something similar to the number 86 printed to the screen, which is the number of properties for `win32_printer`.

To leverage the `get-cimclass` cmdlet to view the class methods, do the following:

```
(get-cimclass -class win32_Printer).CimClassMethods
```

The output of this command is shown in the following screenshot:

Name	ReturnType	Parameters	Qualifiers
SetPowerState	UInt32	{PowerState, Time}	{}
Reset	UInt32	{}	{Description, Implemented,...}
Pause	UInt32	{}	{Description, Implemented,...}
Resume	UInt32	{}	{Description, Implemented,...}
CancelAllJobs	UInt32	{}	{Description, Implemented,...}
AddPrinterConnection	UInt32	{Name}	{Description, Implemented,...}
RenamePrinter	UInt32	{NewPrinterName}	{Description, Implemented,...}
PrintTestPage	UInt32	{}	{Description, Implemented,...}
SetDefaultPrinter	UInt32	{}	{Description, Implemented,...}
GetSecurityDescriptor	UInt32	{Descriptor}	{description, implemented,...}
SetSecurityDescriptor	UInt32	{Descriptor}	{description, implemented,...}

This example displays the proper syntax for retrieving the `CimClassMethods` property of the `win32_printer` class. You first enclose the command in parentheses, and call `get-cimclass` with the `-class` parameter pointing to `win32_printer`. Then, leverage the dot notation to call the `CimClassMethods` property of that item. After executing the command, you will see all the methods, or actions that you can perform on that class.

Tip

An alternative method to write this code is leveraging the pipe command and not using the dot notation. The PowerShell code would look similar to `$method = get-cimclass -class win32_Printer | foreach-object CimClassMethods`. You may also pipe the output to the selection criteria of `select -ExpandProperty CimClassMethods`. The PowerShell code would look similar to `$method = get-cimclass -class win32_Printer | select -ExpandProperty CimClassMethods`.

To search for method qualifiers using the `get-cimclass` cmdlet, you can perform the following:

```
$method = (get-cimclass -class win32_Printer) .CimClassMethods |  
where {$_.name -eq "SetDefaultPrinter"}  
$method  
$method.qualifiers
```

The output of this set of commands is shown in the following screenshot:

```

PS C:\> $method = (get-cimclass -class win32_Printer).CimClassMethods | where {$_.Name -eq "SetDefaultPrinter"}
PS C:\> $method

Name                                     ReturnType Parameters
----                                     -----  -----
SetDefaultPrinter                         UInt32  {}

PS C:\> $method.Qualifiers

Name          Value
----          -----
Description   The SetDefaultPrinter meth...
Implemented   True
ValueMap     {0, ...}
Values       {Success, Other}

Qualifiers

CimType          Flags
----          -----
String  ..., ToSubclass, Translatable
Boolean  EnableOverride, Restricted
StringArray  EnableOverride, ToSubclass
StringArray  ..., ToSubclass, Translatable

```

The example we just saw displays how you can dig deeper into the `win32_printer` class on a system. You first start by declaring the variable of `$method` and set it equal to the output from the next command. Next, you enclose the `get-cimclass` cmdlet with the `-class` parameter pointing to `win32_printer`. Then you use the dot notation to call the `CimClassMethods` for that item. From the previous example, you know that one of the `Method` properties of `win32_printer` is the value `Name`. You also know that one of the items is `SetDefaultPrinter`. To view more details about the method, you then pipe the output of that command to the evaluation statement of `where {$_.Name -eq "SetDefaultPrinter"}`. On executing the first line, you print to screen the `SetDefaultPrinter` information by calling the variable, `$method`. You then discover one of the properties of the `SetDefaultPrinter` method as `Qualifiers`. You leverage the dot notation to view the `Qualifiers` property by calling `$method.Qualifiers`. After executing this command, the properties of that property will print to the screen.

Creating, modifying, and removing WMI property instances

PowerShell provides the ability to create, modify, and remove new properties in WMI classes. If you want to modify an instance of a property, you have to determine if the property has writeable attributes using the `get-cimclass` cmdlet. To do this, you select a WMI class by calling the `get-cimclass` cmdlet and referencing the class you want to evaluate. You then gather the expanded properties of the class by piping the `get-cimclass` output to the selection criteria of `Select -ExpandedProperty CimClassProperties`.

After gathering the expanded properties, the results need to be piped to the selection criteria of `where {$__.Qualifiers -match "write"}`. On entering this command, you will see all the properties that permit writing and removing properties. Subsequently, if you want to see the properties that are read-only, you can change the selection criteria of `where {$__.Qualifiers -notmatch "write"}`. This will display just the read-only properties.

To determine the writeable properties for the `win32_environment` class, do the following:

```
Get-cimclass win32_Environment | select -ExpandProperty  
CimClassProperties | where {$__.Qualifiers -match "write"}
```

The output of these commands is shown in the following screenshot:

```
PS C:\> Get-cimclass win32_Environment | select -ExpandProperty CimClassProperties | where {$_.Qualifiers -match "write"}  
  
Name : Name  
Value :  
CimType : String  
Flags : Property, Key, NullValue  
Qualifiers : {read, key, MappingStrings, Override...}  
ReferenceClassName :  
  
Name : VariableValue  
Value :  
CimType : String  
Flags : Property, NullValue  
Qualifiers : {MappingStrings, read, write}  
ReferenceClassName :
```

This example displays how to use the `get-cimclass` cmdlet and the selection criteria to determine what properties in a class have the 'write' qualifier. You first start by calling the `get-cimclass` cmdlet referencing the `win32_environment` class. Next, you pipe those results to the selection criteria of `select -ExpandProperty CimClassProperties`. These results are finally piped to the selection criteria of `where {$_.Qualifiers -match "Write"}`. This will output to the console all the properties that have the 'write' qualifier.

To determine the non-writeable properties for the `win32_environment` class, do the following:

```
Get-cimclass win32_Environment | select -ExpandProperty CimClassProperties | where {$_.Qualifiers -notmatch "write"} | select -ExpandProperty Name
```

The following screenshot displays the output of these commands:

```
PS C:\> Get-cimclass win32_Environment | select -ExpandProperty CimClassProperties | where {$_.Qualifiers -notmatch "write"} | select -ExpandProperty Name  
Caption  
Description  
InstallDate  
Name  
Status  
SystemVariable  
UserName  
VariableValue
```

This example displays how to use the `get-cimclass` cmdlet, and the selection criteria to determine the properties in a class that do not have the 'write' qualifier. You first start by calling the `get-cimclass` cmdlet referencing the `win32_environment` class. Then, you pipe those results to

the selection criteria of `select -ExpandProperty CimClassProperties`. These results are piped to the selection criteria of `where {$_.Qualifiers -notmatch "Write"}`. You finally pipe those results to the selection criteria of `select -ExpandProperty Name`. The final pipe that you follow in the sequence is to truncate the list of items. If you didn't select only the `Name` property, all the properties of `CimClassProperties` would print to the screen. After executing the command, the `Name` property of `Qualifiers` which do not have the 'write' property will be printed to the console.

Once you find a class that has writeable properties, you need to determine what property values are required for that particular class. You can do this by using the `get-ciminstance` cmdlet, and declaring a class name to find the required fields.

To use the `get-ciminstance` cmdlet with the `win32_environment` class, do the following:

```
get-ciminstance win32_environment
```

The output of this command is shown in the following screenshot:

PS C:\> get-ciminstance win32_environment		
Name	UserName	VariableValue
ComSpec	<SYSTEM>	%SystemRoot%\system32\cmd.exe
FP_NO_HOST_CHECK	<SYSTEM>	NO
OS	<SYSTEM>	Windows_NT
Path	<SYSTEM>	C:\ProgramData\Oracle\Java\javapath;C:\ProgramCOM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WS...
PATHEXT	<SYSTEM>	AMD64
PROCESSOR_ARC...	<SYSTEM>	%SystemRoot%\TEMP
TEMP	<SYSTEM>	%SystemRoot%\TEMP
TMP	<SYSTEM>	SYSTEM
USERNAME	<SYSTEM>	

This example displays how to use the `get-ciminstance` cmdlet to determine the properties that need to be created for new properties in the `Win32_Environment` class. You first start by running the `get-ciminstance` cmdlet, pointing to `Win32_Environment`. After running this script, you will see that there are three properties for each property. These properties include `Name`, `UserName`, and `VariableValue`. In the instance that you want to create a new instance property in the `win32_environment` class, the properties of `Name`, `UserName`, and

`VariableValue` would be required.

Creating property instances

To create a new instance of a class property, you can leverage the `new-ciminstance` cmdlet. The proper syntax of using this cmdlet is calling `new-ciminstance`, and referencing the class that you want to create a new property in. You will then leverage the `-property` parameter and create a hash table of items that are required for the new property. This is done by creating a hash table similar to

`@{Property="SomeValueName";Property="Value You Want To Set The property To"}`. When you create a property, WMI validates the input of these values prior to setting it in the WMI infrastructure. If you are missing properties or they don't meet a certain criteria, the command will fail.

Note

The `UserName` property in the `Win32_Environment` class is validated against the **Security Account Manager (SAM)** of a Windows system. If the username specified in the `UserName` property isn't a valid user on the computer, this script will fail.

In the instance where you want to add a new property to the `win32_environment` class, do the following:

```
# Update the Domain\Username with valid credentials
New-CimInstance Win32_Environment -Property
@{Name="PurchasedDate";VariableValue=
"10/17/2015"; UserName="DOMAIN\USERNAME"}
Get-Ciminstance Win32_Environment | Where {$_.name -match
"PurchasedDate"}
```

The output of this set of commands is shown in the following screenshot:

```

PS C:\> # Update the Domain\Username with valid credentials
PS C:\> New-CimInstance Win32_Environment -Property @{"Name="PurchasedDate";VariableValue="10/17/2015"; UserName="DOMAIN\USERNAME"};
Name          UserName          VariableValue
----          -----          -----
PurchasedDate DOMAIN\USERNAME          10/17/2015

PS C:\> Get-Ciminstance Win32_Environment | Where {$_.name -match "PurchasedDate"}
Name          UserName          VariableValue
----          -----          -----
PurchasedDate DOMAIN\USERNAME          10/17/2015

```

This example displays how to successfully create a new property with three properties in the `Win32_Environment` class. After running the previous example, you learned that the three required properties for the `Win32_Environment` class are `Name`, `UserName`, and `VariableName`. You first start by leveraging the `New-CimInstance` cmdlet pointing to the `Win32_Environment` class, and the `-Property` parameter pointing to a hash table of objects. The hash table you build is

`@{ Name="PurchasedDate";VariableValue="10/17/2015";
UserName="DOMAIN\USERNAME"}`. After executing this script, there will be a new property in `Win32_Environment` with the property of `Name` set to the value of `PurchasedDate`, the property of `UserName` set to the value of `DOMAIN\USERNAME`, and the property of `VariableValue` set to the value of `10/17/2015`.

After creating the new property, you then leverage the `Get-CimInstance` cmdlet pointing to the `Win32_Environment` class to validate the creation. You pipe the output to the selection criteria of `Where {$_.Name -Match "PurchasedDate"}`. After submitting this command, the console will print the property with all its properties.

Modifying property instances

If you want to modify instances of properties, you need to use both the `Get-CimInstance` and the `Set-CimInstance` cmdlets. This is due to you needing to place the WMI object into a variable to modify it and set it back into the WMI infrastructure. To start, you first have to declare a variable for the WMI instance and set it equal to the `Get-CimInstance` cmdlet with the `-Class` parameter pointing to a class. This will access an

instance of the class and store it in the variable that you declared. You then will use the `Set-CimInstance` cmdlet to modify the property. To do this, you will call `Set-CimInstance` with the `-CimInstance` parameter pointing to the variable you defined earlier. You then declare the `-Property` parameter and build a hash table of what you want to set it to. The hash table will look similar to

`@{Name="SomePropertyName";VariableValue="Value You Want To Set It To"}`. When you modify a property, WMI validates the input of these values prior to setting it in the WMI infrastructure. After validating all of the properties, you will have successfully updated that property with the `Set-CimInstance` cmdlet.

To modify a property in the `Win32_Environment` class, do the following:

```
$instance = Get-CimInstance Win32_Environment | Where {$_.name -match "PurchasedDate"}  
Set-CimInstance -CimInstance $instance -Property  
@{Name="PurchasedDate";VariableValue="October 17, 2015";}  
Get-CimInstance Win32_Environment | Where {$_.name -match "PurchasedDate"}
```

The output of these commands is shown in the following screenshot:

```
PS C:\> $instance = Get-CimInstance Win32_Environment | Where {$_.name -match "PurchasedDate"}  
PS C:\> Set-CimInstance -CimInstance $instance -Property @{Name="PurchasedDate";VariableValue="October 17, 2015";}  
PS C:\> Get-CimInstance Win32_Environment | Where {$_.name -match "PurchasedDate"}  


| Name          | UserName        | VariableValue    |
|---------------|-----------------|------------------|
| PurchasedDate | DOMAIN\USERNAME | October 17, 2015 |


```

This example displays how you can use the `Set-CimInstance` cmdlet to change properties of a WMI property. You first start by using the `Get-CimInstance` cmdlet with the `Win32_Environment` class piped to the selection criteria of `Where {$_.Name -match "PurchasedDate"}`. You then store this WMI property object in a variable named `$instance`. You then continue to use the `Set-CimInstance` cmdlet with the `-CimInstance` parameter pointing to the `$instance` variable, and the `-Property` parameter with a new hash table of objects. The hash table you specify is `@{Name="PurchasedDate";VariableValue="October 17, 2015";}`, which updates the property with the Name of `PurchasedDate` and

property the `VariableValue` to reflect October 17, 2015. You then use the `get-ciminstance` cmdlet pointing to the `Win32_Environment` class with the selection criteria of `where {$_.Name -match "PurchasedDate"}`. After executing this command, the console will print the updated property of the `Win32_Environment` class for the `Name` value of `PurchasedDate`. You will see that the `VariableValue` property has been updated to October 17, 2015.

Tip

In this example, the `UserName` property is a read-only property. If you attempt to update or use this value, you will receive an error message. In this example, you will only call the `Name` and `VariableValue` properties in the array you are building, to successfully update that WMI property.

There are properties, however, which, after creation, become Read-Only. Even though you were able to define them during the creation process, the class may prevent these properties from undergoing future modification. The only way to change these properties would be to remove the property, and recreate it with different properties. To determine if a class has 'write' properties, you can leverage the `get-cimclass` cmdlet to evaluate the `Quantifier` properties that permit writing.

Removing property instances

If you want to remove instances of properties, you need to use both the `get-ciminstance` and the `remove-ciminstance` cmdlets. To start with, you have to declare a variable for the instance and set it equal to the `get-ciminstance` cmdlet, with the `-class` parameter pointing to a class. You then need to pipe it to the selection criteria of `where {$_.Name -like "PropertyName"}`. This will access an instance of the property and store it in the variable that you declared. The second step is to remove the property by using the `remove-ciminstance` cmdlet. The proper syntax of using this cmdlet is calling `remove-ciminstance`, and the `-ciminstance` parameter pointing to the variable you defined earlier. After executing this step, the property will be removed from the system.

To remove an instance of a property in the `win32_environment` class, do the following:

```
$instance = Get-Ciminstance win32_environment | Where {$_.name -match "PurchasedDate"}  
Remove-ciminstance -ciminstance $instance  
Get-Ciminstance win32_environment | Where {$_.name -match "PurchasedDate"}
```

This set of commands gives the output as shown in the following screenshot:

```
PS C:\> $instance = Get-Ciminstance win32_environment | Where {$_.name -match "PurchasedDate"}  
PS C:\> Remove-ciminstance -ciminstance $instance  
PS C:\> Get-Ciminstance win32_environment | Where {$_.name -match "PurchasedDate"}
```

This example displays how to delete a WMI property leveraging the `remove-ciminstance` cmdlet. You first start by using the `get-ciminstance` cmdlet, with the `Win32_Environment` class piped to the selection criteria of `where {$_.name -match "PurchasedDate"}`. You then store this WMI property object in a variable named `$instance`. Next, you continue to use the `remove-ciminstance` cmdlet with the `-ciminstance` parameter pointing to the `$instance` variable. On executing this command, the WMI property is deleted from the system. You verify this by using the `get-ciminstance` cmdlet pointing to the `Win32_Environment` class with the selection criteria of `where {$_.name -match "PurchasedDate"}`. You will not receive any results from this, implying, therefore, that the instance of the property has been removed.

Invoking WMI class methods

WMI methods enable you to execute different activities on a system. PowerShell provides the ability to hook onto these methods to perform different actions using the `invoke-cimmethod` cmdlet. In order to determine what methods are available for use in a WMI class, you can leverage the `get-cimclass` cmdlet, with the optional `-class` parameter pointing to a WMI class. You then can pipe those results to the selection criteria of `| select -ExpandProperty CimClassMethods`. This will display all the methods and properties for those methods in that WMI class. This will help you expose what methods are available for a particular class.

To leverage the `get-cimclass` cmdlet to see the methods in the `win32_process` class, you can do the following:

```
get-cimclass win32_process | select -ExpandProperty  
CimClassMethods
```

The output of this command is shown in the following screenshot:

Name	ReturnType	Parameters	Qualifiers
Create	UInt32	{CommandLine, CurrentDirec...}	{Constructor, Implemented,...
Terminate	UInt32	{Reason}	{Destructor, Implemented,...
GetOwner	UInt32	{Domain, User}	{Implemented, MappingStrin...
GetOwnerSid	UInt32	{Sid}	{Implemented, MappingStrin...
SetPriority	UInt32	{Priority}	{Implemented, MappingStrin...
AttachDebugger	UInt32	{}	{Implemented, ValueMap}

There are two popular ways to utilize the `invoke-cimmethod` cmdlet. The first is to use the `-MethodName` parameter referencing a method name, along with the `-arguments` parameter with a hash table of options. The hash table for the arguments would look similar to

```
@{Property="ActionItem"}.
```

To leverage the `invoke-cimmethod` cmdlet with `-arguments` parameter, do the following:

```
Invoke-CimMethod Win32_Process -MethodName "Create" -Arguments
```

```
@{ CommandLine = 'mspaint.exe' }
```

The output of this command is shown in the following screenshot:

PS C:\> Invoke-CimMethod Win32_Process -MethodName "Create" -Arguments @{ CommandLine = 'mspaint.exe' }		
ProcessId	ReturnValue	PSComputerName
9224	0	

This example displays how to use the `Invoke-CimMethod` cmdlet with the `-Arguments` parameter to start a new process for Microsoft Paint, using the `Win32_Process` class. You first start by leveraging the `Invoke-CimMethod` cmdlet calling the `Win32_Process` class. You then use the `-MethodName` parameter with the `Create` parameter, and the `-Arguments` parameter with the array of `@{ CommandLine = 'mspaint.exe' }`. On executing this command, the method will create a new `ProcessId` and launch the `mspaint.exe` application.

The second method to use the `Invoke-CimMethod` cmdlet is through using a query parameter. The proper syntax of using the query parameter is calling `Invoke-CimMethod` referencing a class name, then the `-MethodName` parameter referencing the method you want to invoke , and finally the `-Query` parameter with a **WMI Query Language (WQL)** query to run against that method name.

To leverage the `Invoke-CimMethod` cmdlet with the `-Query` parameter, do the following:

```
Invoke-CimMethod -Query 'select * from Win32_Process where name like "mspaint.exe"' -MethodName "Terminate"
```

The output of this command is shown in the following screenshot:

PS C:\> Invoke-CimMethod -Query 'select * from Win32_Process where name like "mspaint.exe"' -MethodName "Terminate"		
ReturnValue	PSComputerName	
0	0	

This example displays how to use the `invoke-cimmethod` cmdlet with the `-query` parameter to terminate the Microsoft Paint process using the `win32_process` class. You first start by leveraging the `invoke-cimmethod` cmdlet calling the `win32_process` class. You then use the `-query` parameter with the WQL query of '`select * from Win32_Process where name like "mspaint.exe"`', and the `-MethodName` parameter with the `Terminate` parameter. On executing this command, the method will terminate all processes that have the "`name`" property similar to `mspaint.exe`. After executing this you will see `ReturnValue` is 0, which means that it has been successful. If you attempt to run this command a second time, there will be no return, due to the application already being terminated.

Summary

This chapter explained how to use PowerShell to interact with WMI. It showed you the components that make up the WMI, namely consumers, infrastructure, and providers. It also explained how these WMI components are used while interacting with the WMI and PowerShell.

You learned a variety of cmdlets that allow you to navigate the WMI structure. You explored how to search different WMI classes for their attributes, which include methods and properties. You then worked through creating, modifying, and removing WMI object property instances. At the end of the chapter, you learned how to invoke WMI class methods in a variety of ways. In the next chapter you will dive into the XML structure and learn how to leverage PowerShell to read and manipulate XML based items.

Chapter 11. XML Manipulation

When you are working with Microsoft-based systems, there is a high probability that you are leveraging **eXtensible Markup Language (XML)** for data and communications. XML was created by the **World Wide Web Consortium (W3C)** to standardize the encoding of documents to make them both legible to humans and usable by computer systems. XML's format is very similar to that of **Hypertext Markup Language (HTML)**. If you know the basics of HTML, you should be able to pick up XML pretty quickly. While the syntax is very similar between HTML and XML, the purposes of these languages are very different. HTML is used by web browsers to render objects and text on a website. XML is used to encapsulate data to be stored on a system, or passed between systems.

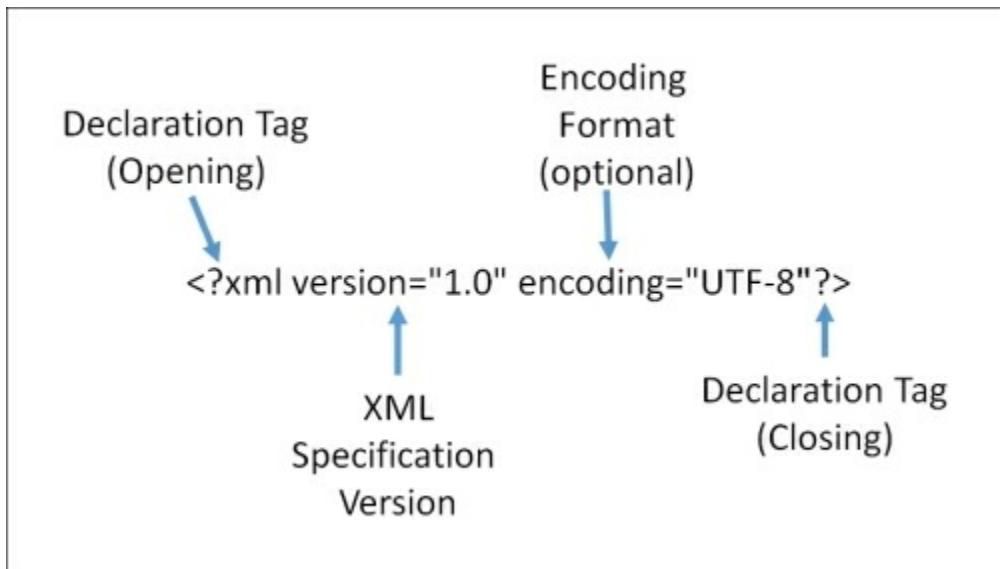
In this chapter, we will learn about:

- The XML file structure
- Reading XML files
- Adding XML content
- Modifying XML content
- Removing XML content

XML file structure

When PowerShell interacts with XML, it leverages an XML reading engine known as an **XML parser**. Much like how PowerShell parses PS1 scripts, the XML parser will read line by line and interpret the contents of the XML file. When the PowerShell XML parser reads the file, it has all of the encoding logic built in, so that it can read the different parts of the XML file. After the XML parser reads the file, it will make the contents available for use within your PowerShell scripts.

For an XML parser to know the file is an XML file, you have to make an XML declaration at the beginning of the file. The following graphic represents a properly created XML declaration:

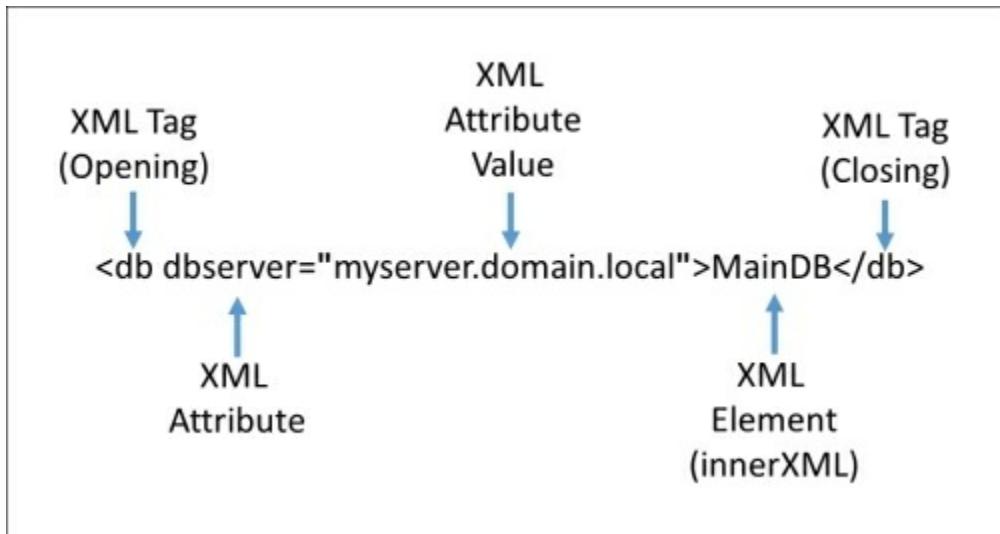


The XML declaration is a mandatory line at the very beginning of the XML document itself. There are several parts of the XML declaration that are mandatory. The declaration tag of XML starts by leveraging the code `<?xml` and is required to tell the XML parser that the version and encoding items may be following. You must then specify the mandatory attribute of `version`, equal to an XML standardization version number. You can then specify the optional `encoding` attribute, where you have the ability to define what format of `encoding` the XML file has been prepared in. To close the declaration section of the code, you use the `?>` closing tag. The preceding example is declaring that an XML file will be XML version 1 with the encoding format set to UTF-8.

Note

UTF-8 and UTF-16 are the two common XML encoding types used with PowerShell. **Universal Characters Set Transformation Format 8-bit/16-bit (UTF-8/16)** is the default Unicode character set that can be used in the XML file. UTF-8 is the most used encoding on the World Wide Web.

After you specify the encoding, you can start defining the data inside the XML file itself. This data is represented as XML tags, which can contain elements (`innerXML`), attributes, and attribute values. The following graphic represents a properly created XML tag:



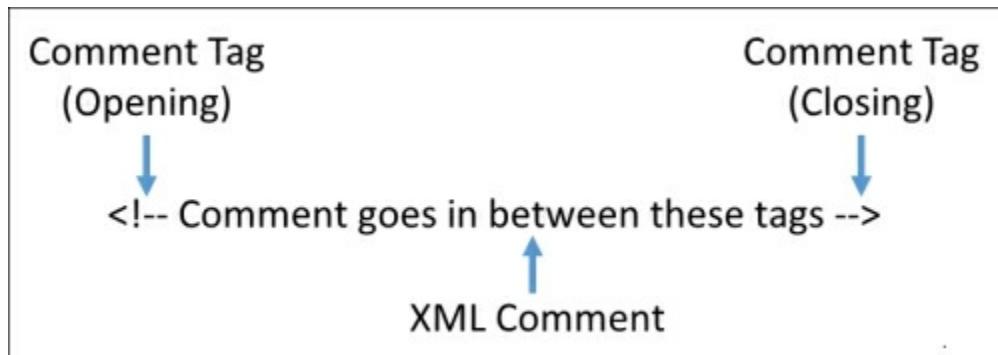
When you define an XML tag, you are declaring an object that contains the related data about that tag. There are two methods to define the data about the individual XML tag. The first method is leveraging attributes and attribute values. An attribute is a unique name that describes an item related to the XML tag. The attribute value is whatever data you want to store for that attribute's value. In the preceding example, the XML tag is for a database or `db`. The attribute refers to a database server name or `dbserver`, and the value is the actual server name of `myserver.domain.local`. You can define multiple attributes and attribute values per tag, which can be helpful in linking similar data together.

The second method is to leverage the elements or the `innerXML` data. This is the data that is also associated with the tag. When the XML parser calls a tag in a script, the element is returned for use. In the preceding example, the XML element is `MainDB`, which is the database name correlated to the `db` tag. Elements can only store one value per tag. This is why many developers skip using the elements and only leverage the attributes, as they provide multiple corresponding data points per tag.

The alternative to multiple attributes is to provide multiple XML tags, and process each individual tag gathering the elements. There are no official rules defined by W3C that specify which you should use per

XML file. It is recommended, however, to only use one format syntax per file.

In instances where you want to make notes about the XML tags, attributes, or elements, you may want to leverage the use of comments. The following graphic represents a properly created XML comment:



Comments are data that are just like comments in PowerShell. They provide developers with information on each of the sections or individual items in an XML file. The comments have the special opening comment tag of <!--. To close the comment, you can leverage the closing comment tag of -->. Any text or XML data in between these two comment tags are ignored by the XML parser on a system.

As you are building multiple tags in your XML files, you will need to follow the XML W3C **Document Object Model (DOM)** tree structure. In simple terms, the XML DOM tree structure consists of a parent, child, and siblings. The parent is a grouping tag that groups the child tags. If there are multiple child tags in the parent tag, the individual child tags are siblings.

An XML file with parents, children, and siblings may look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings> <!--Parent Tag-->
    <db dbserver="myserver.domain.local">MainDB</db> <!--Child
Tag-->
</settings>
<access> <!-- Parent Tag -->
```

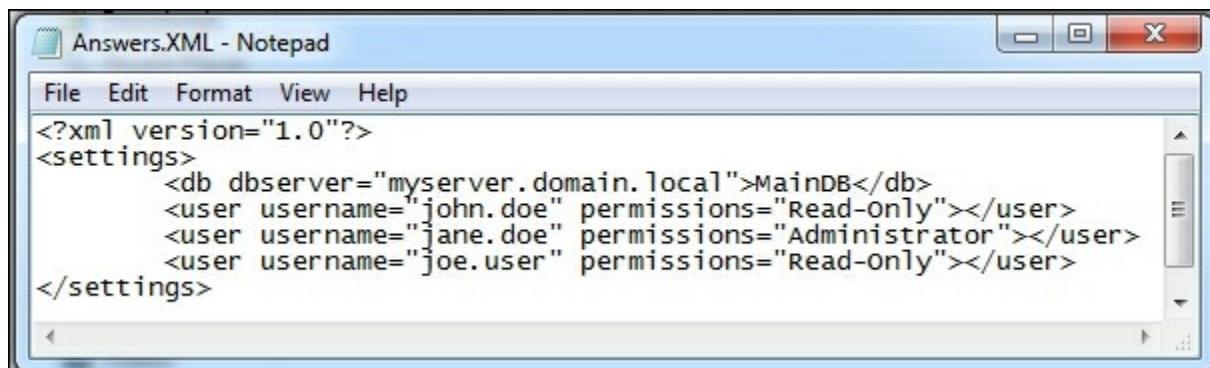
```
<user>user1</user> <!-- Child and Sibling Tag -->
<user>user2</user> <!-- Child and Sibling Tag -->
<group>Group1</group> <!-- Child and Sibling Tag -->
</access>
```

The XML structure is a hierarchy of components within each other. The preceding example displays the contents of an XML file with multiple parent, child, and sibling tags. Each tag can have child tags which can be named the same, like the `<user>` tags, to represent a grouping of similar child elements. You also define a child tag like `<group>`, which specifies a child and a sibling tag that has different elements.

Reading XML files

PowerShell has the ability to natively read and parse the data in XML files. This is done by loading an XML file into an XML document object leveraging the `get-content` cmdlet. You start by defining a variable like `$xmlfile` with the data type set to `[xml]`. You then call the `get-content` cmdlet with the `-path` argument set to the location of an XML file. If you omit the `[xml]` data type while defining the variable, it will interpret the file as a text file. This means that you will not be able to leverage any of the built-in XML support.

The following graphic represents the required XML file to complete the remaining examples in this chapter:



A screenshot of a Windows Notepad window titled "Answers.XML - Notepad". The window contains the following XML code:

```
<?xml version="1.0"?>
<settings>
    <db dbserver="myserver.domain.local">MainDB</db>
    <user username="john.doe" permissions="Read-Only"></user>
    <user username="jane.doe" permissions="Administrator"></user>
    <user username="joe.user" permissions="Read-Only"></user>
</settings>
```

To complete the examples in this chapter, you will need to create a new XML file named `Answers.xml` in the location of `c:\Program`

Files\MyCustomSoftware2\. Leveraging the text editor of your choice, insert the preceding data inside the Answers.xml file. The examples also build on each other, so you will want to execute this chapter sequentially to have all of the code samples work appropriately.

Tip

You may also refer to this chapter's code file to quickly create the described content.

After creating the example (XML file), you can load the Answers.xml file into a variable:

```
$xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
$xml = [xml] (get-content $xmlfile)
$xml
```

The output of this script is shown in the following screenshot:

```
PS C:\> $xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
PS C:\> $xml = [xml] (get-content $xmlfile)
PS C:\> $xml
xml
---
version="1.0"                                     settings
-----                                         settings
```

The preceding example displays how to properly load an XML document into an XML document object by leveraging the get-content cmdlet. You start by defining the \$xmlfile variable and setting it equal to c:\Program Files\MyCustomSoftware2\Answers.xml. You then define the \$xml variable, leverage the get-content cmdlet pointing to the \$xmlfile and defining a data type of [XML]. You then call the \$xml variable to see its contents. You will see that the contents are the DOM tree structure of the XML file you created.

After you load the contents of an XML file into memory, you can interact with the data, using different methods. One method to navigate the XML document object is to leverage the dot notation. By using the dot notation, you can retrieve all of the parent, child, and sibling tags

simply by calling the tag names.

To use the dot notation to navigate the XML file, do the following action:

```
$xml.xml  
$xml.settings  
$xml.settings.db  
$xml.settings.user
```

The output of this script is shown in the following screenshot:

The screenshot shows a PowerShell window with the following command history and output:

```
PS C:\> $xml.xml
version="1.0"
PS C:\> $xml.settings
db
-- 
db
          user
          ----
          {user, user, user}

PS C:\> $xml.settings.db
dbserver
-----
myserver.domain.local
          #text
          -----
          MainDB

PS C:\> $xml.settings.user
username
-----
john.doe
jane.doe
joe.user
          permissions
          -----
          Read-Only
          Administrator
          Read-Only
```

This example displays how to navigate the XML file by using the dot notation. You start by referencing the XML declaration tag by typing `$xml.xml`. You will see the version attribute printed to the screen. You then view the parent tag by typing `$xml.settings`. The output of this command displays the two child objects of `db` and `user`. You then use the dot notation to view the `db` child object by typing `$xml.settings.db`. You will see the attributes and elements of the child object `db`. Last, you view the multiple siblings of the tag `user` by typing `$xml.settings.user`. The siblings and their attributes print to the console after executing the command.

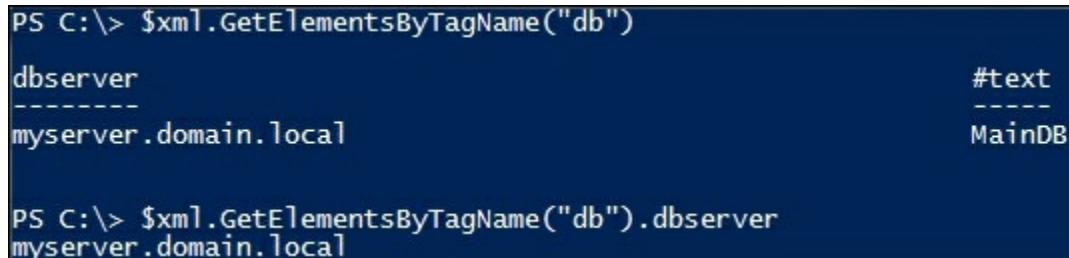
Another approach to navigating an XML file is through the use of XML methods. The `.GetElementsByTagName()` method enables you to search the XML file for tags named as specific values. This will return the attributes and elements of an XML tag. In the instance when there are tags that are named the same, the method will return all values that are equal to the tag value specified in the method. This method provides data in a format where you can leverage the dot notation to obtain the attributes and elements of the tag. This is done by calling

`$XMLVariable.GetElementsByTagName("Tag").AttributeName` for an attribute and `$XMLVariable.GetElementsByTagName("Tag").#text` for the element data.

To retrieve attributes leveraging the `.GetElementsByTagName()` method, do the following operation:

```
$xml.GetElementsByTagName("db")
$xml.GetElementsByTagName("db").dbserver
```

The output of this script is shown in the following screenshot:



```
PS C:\> $xml.GetElementsByTagName("db")
dbserver
-----
myserver.domain.local
#text
-----
MainDB

PS C:\> $xml.GetElementsByTagName("db").dbserver
myserver.domain.local
```

This example displays how to read the `dbserver` attribute of the `db` tag by using the `.GetElementsByTagName()` method and dot notation. You start by viewing the `db` tag by leveraging the `GetElementsByTagName` method with the value of `db`. You see that `db` has two properties the `dbserver` attribute and the `#text` node type. You then use the dot notation to view the attribute value inside the `dbserver` attribute by calling `$xml.GetElementsByTagName("db").dbserver`. The console will print to the screen `myserver.domain.local`, which is the attribute value for `dbserver` in the `db` tag.

To retrieve the elements leveraging the `.GetElementsByTagName()` method, do the following operation:

```
$xml.GetElementsByTagName("db")
$xml.GetElementsByTagName("db")."#text"
```

The output of this command is shown in the following screenshot:

```
PS C:\> $xml.GetElementsByTagName("db")
dbserver
-----
myserver.domain.local
```

	#text
-----	-----
	MainDB

```
PS C:\> $xml.GetElementsByTagName("db")."#text"
MainDB
```

This example displays how to read the text element in the `db` tag. You start viewing the `db` tag by leveraging the `GetElementsByTagName` method with the value of `db`. You see that `db` has two properties, the `dbserver` attribute and the `#text` node type. You then use the dot notation to view the attribute value inside the `#text` node type by calling `$xml.GetElementsByTagName("db")."#text"`. The console will print to the screen `MainDB`, which is the element in the `db` tag.

To view the individual sibling `user` tags and print their attribute values to the console, do the following action:

```
$users = $xml.GetElementsByTagName("user")
Foreach ($user in $users) {
    Write-host "Username: " $user.username
    Write-host "Permission: " $user.permissions
    Write-host ""}
```

The output of this script is shown in the following screenshot:

```
PS C:\> $users = $xml.GetElementsByTagName("user")
PS C:\> Foreach ($user in $users) {
>> Write-host "Username: " $user.username
>> Write-host "Permission: " $user.permissions
>> Write-host ""
>>
>>
Username: john.doe
Permission: Read-Only

Username: jane.doe
Permission: Administrator

Username: joe.user
Permission: Read-Only
```

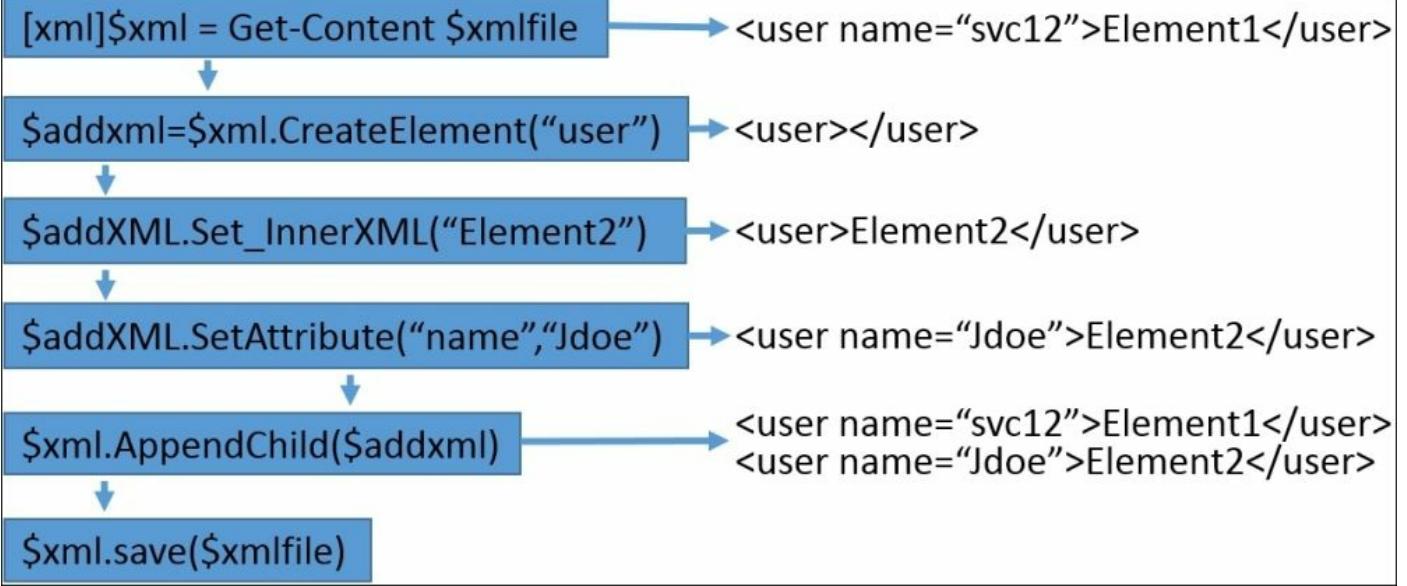
This example displays how to process XML files with tags that are the same. To start, you retrieve the `user` tag data from the XML variable and set it to the `$users` variable. To do this, you create the `$users` variable and set it equal to the result from

`$xml.GetElementsByTagName("user")`. All of the tags that have the name of `user` are now stored in an array in the `$users` variable. You then create a `Foreach` loop to loop through each `$user` in the variable `$users`. You then use the dot notation to write to the console the word `Username:` with the `$user.username` attribute value. You also write to the console the word `Permission:` with the `$user.permissions` attribute values. After executing this script, you will see three users and three permissions that match the values of the XML file.

Adding XML content

When you are working with XML files, there may be instances where you need to add, modify, and remove content from the XML file. PowerShell's integration with XML provides many methods that you can use to manipulate the XML data.

The following example displays how you can add different types of data into an XML file:



Adding XML data to an existing XML structure is a multistep process. This can be done through the following steps:

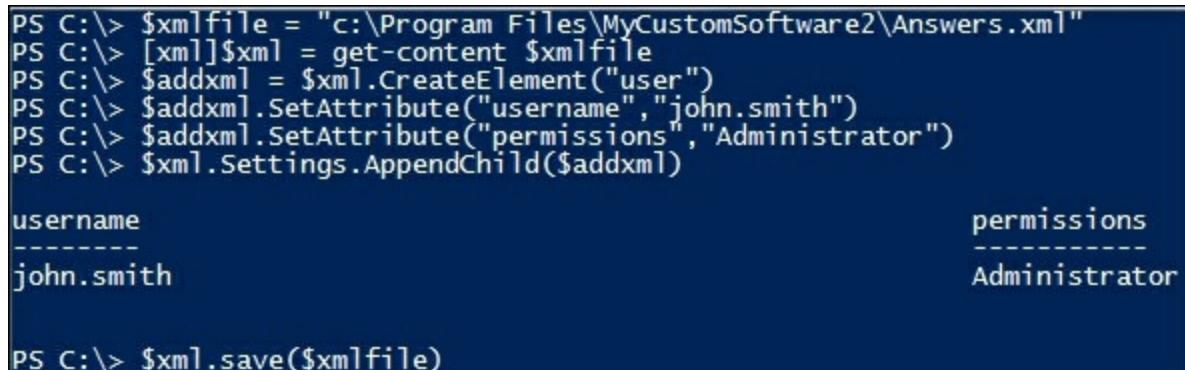
- 1. Retrieving the XML file and place in memory:** As shown in the preceding example, you first have to retrieve the XML document and place it into a variable.
- 2. Creating a new variable containing the new tag object:** You need to create the new user tag object by leveraging `.CreateElement` referencing a new or existing tag name and set the output to a variable. This will create a new instance of the tag to modify the element and attributes.
- 3. Appending elements (if needed):** To set an element to the `user` tag, you can leverage `.Set_innerXML` with the element data you want to add to the XML file.
- 4. Appending attributes and attribute values (if needed):** If you want to add attributes to the `user` tag, you can use `.SetAttribute` with the attribute name and attribute value.
- 5. Merging the new tag, elements, and attributes into the XML content in memory:** At this point, you only created a new `user` tag object in memory, and you need to merge it into the XML document object you have in memory by using `.AppendChild()`, referencing the variable containing the new XML tag.
- 6. Saving updated XML content in memory over the existing XML**

file: When you are done merging the new tag into the XML content in memory, you can save the changes over the existing XML file. When you are ready to save the XML variable back to the file, you can leverage the `.Save()` method, pointing to the XML file path location.

To create a new `user` tag in an existing XML file with elements and attributes, do the following action:

```
$xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
$xml = get-content $xmlfile
$addxml = $xml.CreateElement("user")
$addxml.SetAttribute("username", "john.smith")
$addxml.SetAttribute("permissions", "Administrator")
$xml.Settings.AppendChild($addxml)
$xml.save($xmlfile)
```

The output of this script is shown in the following screenshot:



```
PS C:\> $xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
PS C:\> [xml]$xml = get-content $xmlfile
PS C:\> $addxml = $xml.CreateElement("user")
PS C:\> $addxml.SetAttribute("username", "john.smith")
PS C:\> $addxml.SetAttribute("permissions", "Administrator")
PS C:\> $xml.Settings.AppendChild($addxml)

username
-----
john.smith

permissions
-----
Administrator

PS C:\> $xml.save($xmlfile)
```

This example displays how to create a new tag with attributes and append them to an XML file. You start by declaring the XML file variable of `$xmlfile` set to the location of the XML file which is `c:\Program Files\MyCustomSoftware2\Answers.xml`. You then declare an `[xml]` variable type by calling `[xml]$xml` and setting the contents from `get-content $xmlfile` to this variable. You then create a new tag of `user` by calling the `$xml.CreateElement("user")` method and placing it in the `$addxml` variable. This allows you to modify the new object through the `$addxml` variable. You then apply the `username` attribute with

the `john.smith` attribute data to that new element by using the `$addxml.SetAttribute("username", "john.smith")` method. You can also apply the `permissions` attribute with `Administrator` to the element by using the `$addxml.SetAttribute("permissions", "administrator")` method.

Finally, you leverage the `.appendChild($addxml)` method to the `$xml` variable and save the contents in the `$xml` variable over the XML file using `$xml.Save($xmlfile)`. After executing this command, you will have created a new user tag, with the `username` attribute containing `john.smith`, the `permissions` attribute containing `Administrator`, and a blank XML element.

To create an entire XML tag with an element, do the following action:

```
$xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
$xml = get-content $xmlfile
$addxml = $xml.CreateElement("webserver")
$addxml.set_InnerXML("MyWebServer.domain.local")
$xml.Settings.AppendChild($addxml)
$xml.save($xmlfile)
```

The output of this script is shown in the following screenshot:

```
PS C:\> $xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
PS C:\> [xml]$xml = get-content $xmlfile
PS C:\> $addxml = $xml.CreateElement("webserver")
PS C:\> $addxml.set_InnerXML("MyWebServer.domain.local")
PS C:\> $xml.Settings.AppendChild($addxml)

#text
-----
MyWebServer.domain.local

PS C:\> $xml.save($xmlfile)
```

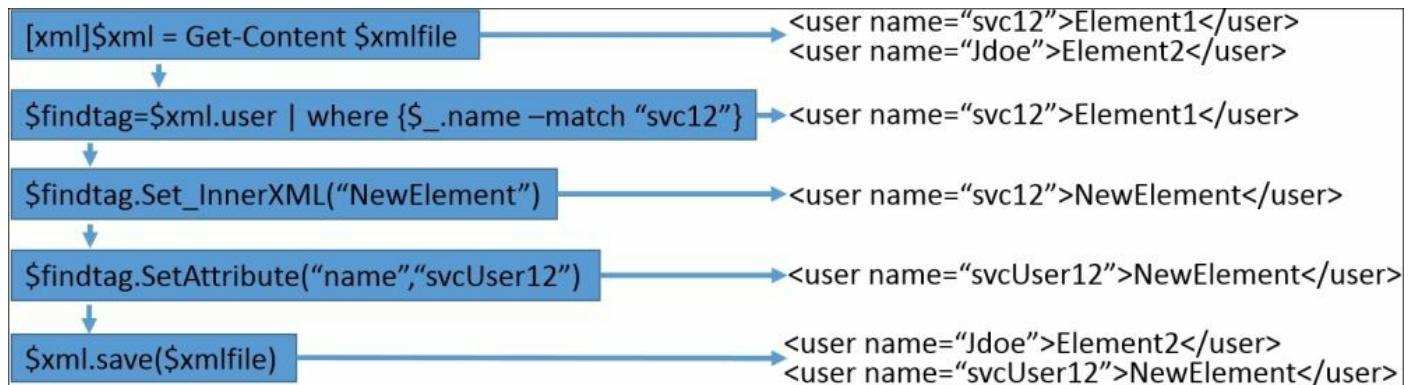
This example displays how to create a new XML tag with elements and append them to an XML file. You start by declaring the XML file variable of `$xmlfile` set to the location of the XML file which is `c:\Program Files\MyCustomSoftware2\Answers.xml`. You then declare an `[xml]` variable type by calling `[xml]$xml` and setting the contents

from `get-content $xmlfile` to this variable. You then create a new tag of webserver by calling the `$xml.CreateElement("webserver")` method and placing it in the `$addxml` variable. This allows you to modify the new object through the `$addxml` variable. You then create the new inner XML element for the webserver tag by declaring

`$addxml.set_innerxml ("MyWebServer.domain.local")`. You append this new child tag and element in the `$xml` variable by declaring `$xml.Settings.AppendChild($addxml)`. Finally, you save the updated content of the `$xml` variable to the XML file using the `$xml.save ($xmlfile)`. After executing this script, `c:\Program Files\MyCustomSoftware2\Answers.xml` will contain a new child tag in the settings tag named webserver with the element of `MyWebServer.Domain.Local`.

Modifying XML content

There may be instances where you need to update content in an XML file. The following graphic displays how you can modify content in an XML file:



The process to modify existing XML tags is similar to adding new XML tags. The main difference is that instead of creating a new element, you search for an existing tag to modify it. To start, you need to use the `get-content` cmdlet to place the contents of the XML file into a variable, named `$xml`. You then declare a variable, like `$findtag` and leverage the dot notation to find the tag you want to modify. If there are multiple tags for the item you are referencing, you can pipe the tag results to the

selection criteria to find an attribute value or element value. In this example, you search for the `name` attribute and use the selection criteria of `where {$_.name -match "svc12"}`.

After obtaining the tag that you want to modify, you can now leverage the `.set_innerXML()` and `.SetAttribute()` methods to modify the tag attributes and elements. After modifying the file, the final step is to leverage the `.Save()` method to append the modified data back into the original XML file.

To search for an attribute and modify the contents and then save it back to a file, do the following action:

```
$xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
$xml = get-content $xmlfile
$findtag = $xml.settings.user | where {$_.username -match
"jane.doe"}
$findtag
$findtag.SetAttribute("permissions", "Read-Only")
$findtag
$xml.save($xmlfile)
```

The output of this script is shown in the following screenshot:

```
PS C:\> $xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
PS C:\> [xml]$xml = get-content $xmlfile
PS C:\> $findtag = $xml.settings.user | where {$_.username -match "jane.doe"}
PS C:\> $findtag
username                                         permissions
-----                                         -----
jane.doe                                         Administrator

PS C:\> $findtag.SetAttribute("permissions", "Read-Only")
PS C:\> $findtag
username                                         permissions
-----                                         -----
jane.doe                                         Read-Only

PS C:\> $xml.save($xmlfile)
```

This example displays how to find a tag, modify attributes, and save it back to an XML file. You start by declaring the XML file variable of

\$xmlfile set to the location of the XML file, which is c:\Program Files\MyCustomSoftware2\Answers.xml. You then declare an [xml] variable type by calling [xml]\$xml and setting the contents from get-content \$xmlfile to this variable. You then declare a variable for the tag you want to update, named \$findtag, and search the \$xml variable by using dot notation to view the \$xml.settings.user tags in the XML file. You then pipe those results to a further selection criteria of where {\$_.username -match "jane.doe"} to search the username attribute for the value of jane.doe. You then print to screen the result contained in \$findtag, which displays the username attribute value of jane.doe and permissions attribute value of Administrator. To modify the permissions attribute, you leverage

\$findtag.SetAttribute("permissions", "Read-Only") to update the tag found in the \$findtag variable. You then print to screen the updated values of \$findtag, where you see that the username attribute value is still jane.doe, but the permissions attribute is now updated to Read-Only. Finally, you save the updated \$xml with the updated tag by leveraging the \$xml.save(\$xmlfile) method. After executing this script, the user tag with the username attribute value set to jane.doe will have the permissions attribute value set to Read-Only.

To search for an element and modify the contents and then save it back to a file, do the following action:

```
$xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
$xml = get-content $xmlfile
$findtag = $xml.settings.db | where {$_.">#text" -match
>MainDB"}
$findtag
$findtag.set_InnerXML("MainDatabase")
$findtag
$xml.save($xmlfile)
```

The output of this command is shown in the following screenshot:

```

PS C:\> $xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
PS C:\> [xml]$xml = get-content $xmlfile
PS C:\> $findtag = $xml.settings.db | where {$_.#text -match "MainDB"}
PS C:\> $findtag

dbserver                               #text
-----                                -----
myserver.domain.local                  MainDB

PS C:\> $findtag.set_InnerXML("MainDatabase")
PS C:\> $findtag

dbserver                               #text
-----                                -----
myserver.domain.local                  MainDatabase

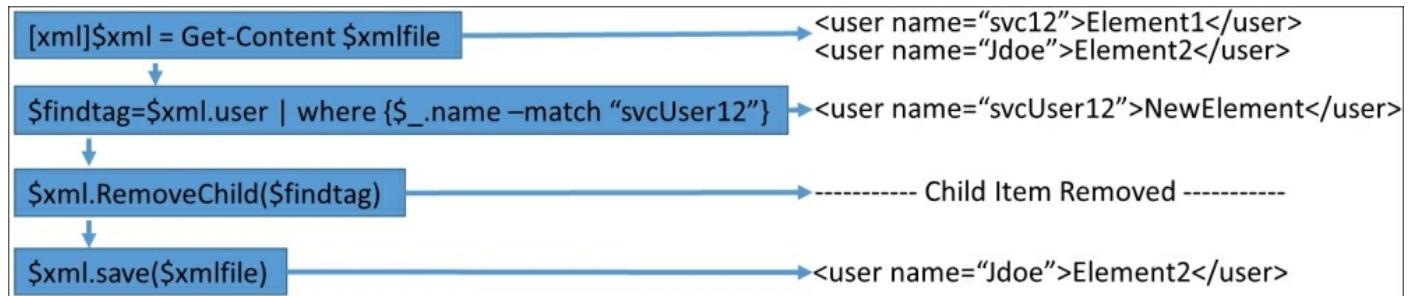
PS C:\> $xml.save($xmlfile)

```

This example shows how to find a tag, modify inner XML, and save it back to an XML file. You start by declaring the XML file variable of `$xmlfile` set to the location of the XML file, which is `c:\Program Files\MyCustomSoftware2\Answers.xml`. You then declare an `[xml]` variable type by calling `[xml]$xml` and setting the contents from `get-content $xmlfile` to this variable. You then declare a variable for the tag you want to update, named `$findtag`, and search the `$xml` variable using dot notation to view the `$xml.settings.db` tags in the XML file. You then pipe those results to a further selection criteria of `where {$_.#text -match "MainDB"}` to search the element for the value of `MainDB`. You then print to screen the result contained in `$findtag`, which displays the `dbserver` attribute value of `myserver.domain.local` and the element value `#text` of `MainDB`. To modify the element of the `db` tag, you then leverage `$findtag.Set_InnerXML ("MainDatabase")` to update the tag found in the `$findtag` variable. You then print to screen the updated values of `$findtag`, where you see that the element value is updated to `MainDatabase`, but the `dbserver` attribute remains `myserver.domain.local`. Finally, you save the updated `$xml` with the updated tag by leveraging the `$xml.save ($xmlfile)` method. After executing this script, the `db` tag with the `dbserver` attribute value of `myserver.domain.local` has an updated element of `MainDatabase`.

Removing XML content

There may be instances where you need to remove content from an XML file. The following graphic displays how you can remove content from an XML file:



When you want to remove XML tags, you can follow a syntax similar to modifying an XML tag. To start, you need to use the `get-content` cmdlet to place the contents of the XML file into a variable named `$XML`. You then declare a variable like `$findtag` and leverage the dot notation to find the tag you want to remove. If there are multiple tags for the item you are referencing, you can pipe the tag results to the selection criteria to find an attribute value or element value. In this example, you search for the name attribute and use the selection criteria of `where {$_ .name -match "svcUser12"}`.

After obtaining the tag that you want to remove, you can now leverage the `.RemoveChild()` methods to remove the tag. After removing the tag from the `$xml` variable, the final step is to leverage the `.Save()` method to append the changes back into the original XML file.

To remove a child item from an XML file, you can perform the following action:

```
$xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
$xml = get-content $xmlfile
$findtag = $xml.settings.user | where {$_.username -match "john.doe"}
$xml.settings.RemoveChild($findtag)
$xml.save($xmlfile)
```

The output of this command is shown in the following screenshot:

```
PS C:\> $xmlfile = "c:\Program Files\MyCustomSoftware2\Answers.xml"
PS C:\> [xml]$xml = get-content $xmlfile
PS C:\> $findtag = $xml.settings.user | where {$_.username -match "john.doe"}
PS C:\> $xml.settings.RemoveChild($findtag)

username                                permissions
-----                                -----
john.doe                               Read-Only

PS C:\> $xml.save($xmlfile)
```

This example displays how to find a tag, remove the tag, and save updates to an XML file. You start by declaring the XML file variable of \$xmlfile set to the location of the XML file, which is c:\Program Files\MyCustomSoftware2\Answers.xml. You then declare an [xml] variable type by calling [xml]\$xml and setting the contents from get-content \$xmlfile to this variable. You then declare a variable for the tag you want to update, named \$findtag, and search the \$xml variable using dot notation to view the \$xml.settings.user tags in the XML file. You then pipe those results to more selection criteria of where {\$_.username -match "john-doe"} to search the username attribute for the value of jane.doe. To remove the user tag with the username attribute value set to john.doe, you leverage the \$xml.settings.RemoveChild(\$findtag) method. PowerShell will print to the screen the tag that has been removed from the \$xml variable. Finally, you save the updated \$xml with the removed tag by leveraging the \$xml.save (\$xmlfile) method. After executing this script, the user tag with the username attribute value set to john.doe will be removed from c:\Program Files\MyCustomSoftware2\Answers.xml.

Summary

This chapter explained how you can leverage PowerShell to manipulate XML files. To start, you learned the core make up of an XML structure. You continued to learn about the DOM tree structure and how parents, children, and siblings relate to each other in an XML file. You also learned how to read XML files and navigate their structure. You learned that you can also use dot notation to navigate an XML structure in PowerShell. Finally, you saw the steps to add, modify, and remove content from an XML file. In the next chapter, you will learn different methods to manage Microsoft systems using PowerShell. You will learn items such as local user and group management, interacting with Windows services, and working with Windows processes.

Chapter 12. Managing Microsoft Systems with PowerShell

This chapter explores many facets of managing Microsoft systems. You will start by learning about the **active directory services interface (ADSI)** adapter, and how it interacts with local objects on a system. You will learn to create and delete users and groups on a local system, and also how to add and remove users from the groups you created. You then learn how to verify that users or groups exist on a system. You will then proceed to learn how to start, stop, and modify Windows services and processes on a system. This chapter ends by explaining how to get information about installed and available Windows features and how to install and remove these features from the system.

Note

This chapter explains interaction with local users and groups. For more information on Active Directory cmdlets, please refer to <https://technet.microsoft.com/en-us/library/ee617195.aspx>.

To properly follow the examples in this chapter, you will need to sequentially execute the examples. Each example builds on the previous examples, and some of them may not function properly if you do not execute the previous steps.

Managing local users and groups

When you are working with building new systems, there may be instances where you need to create new users and groups on a system. While you may be familiar with the `net` commands to manipulate the local users and groups, PowerShell provides other options for communicating with the local system. This section displays how to leverage ADSI calls to interact with local users and groups.

The first step in leveraging ADSI to communicate with a local or remote

system is to set up the connection string. This is done by calling the string [ADSI]"WinNT://SystemName" and storing the call in a variable such as \$ADSI. After the connection is made, you can execute different methods and interact with local objects of that system.

Managing local users

To create a user leveraging the ADSI connection, you first have to declare the connection by calling [ADSI]"WinNT://SystemName" and setting it to the \$ADSI variable. You then leverage the .create() method of the connection variable \$ADSI. Next, you define a new variable for the user, such as \$username, and set it equal to the object output of the .create() method with the arguments of ("User", "User Name"). You then need to define a password for the user, specifying the .setpassword() method on \$username with ("Password") as the argument. Finally, you can commit the user to the **Security Account Manager (SAM)** by calling \$username with the .setinfo() method. After creating the security account, you can add a description for the user by using the \$username.description method and setting it equal to the description you desire. You then have to call the \$username.setinfo() method to commit the change to the SAM.

To create new local users, do the following:

```
$computername = [system.net.dns]::GetHostName()
Function create-user { param($Computer, $username, $password)
    $ADSI = [ADSI]"WinNT://$Computer"
    $user = $ADSI.Create("User", $username)
    $user.setpassword("$Password")
    $user.setinfo()
}
create-user $computername "svcLocalAccount" "P@ssw0rd"
create-user $computername "remLocalAccount" "P@ssw0rd"
```

The output of this is shown in the following screenshot:

```
PS C:\> $computername = [system.net.dns]::GetHostName()
PS C:\> Function create-user { param($Computer, $username, $password)
>> $ADSI = [ADSI]"WinNT://$Computer"
>> $user = $ADSI.Create("User", $username)
>> $user.setpassword("$Password")
>> $user.setinfo()
>> }
>> create-user $computername "svcLocalAccount" "P@sswOrd"
>> create-user $computername "remLocalAccount" "P@sswOrd"
>>
PS C:\>
```

This script displays how to create a function to create local users on a system. You start by defining the computer name by leveraging the `[system.net.dns]` .NET assembly, using its `GetHostName()` method, and saving the name to the `$computername` variable. You then create a function named `create-user` with the `param` arguments of `$computer`, `$username`, and `$password`. Thereafter, you create the `ADSI` connection string into the computer by calling `[ADSI]"WinNT://$computer"` and setting it to the variable `$ADSI`. Then, you invoke the `$ADSI.Create()` method with the arguments of `user` and `$username` to create the object type of `User` with the name of `$username`, and set it to the `$user` variable. Next, you call the `$user.setpassword()` method and specify the password variable of `$password`. To complete the function, you call the `$user.setinfo()` method to set the information into the SAM. You complete the script by calling the `create-user` function with the `$computername` variable, the `svcLocalAccount` username, and the `P@ssword` password. This creates a new user named `svcLocalAccount` with the password of `P@ssword`. You then call the `create-user` function again, with the `$computername` variable, `remLocalAccount` username, and the `P@ssword` password. This creates a new user named `remLocalAccount` with the password of `P@ssword`.

To delete a user, you leverage a similar methodology as creating a user. You first have to declare the connection by calling `[ADSI]"WinNT://SystemName"` and setting it to the `$ADSI` variable. You then leverage the `.Delete()` method of the connection variable `$ADSI` and reference the arguments of `("User", "User Name")`. After running the `$ADSI.Delete("User", "User Name")` method, the User will be deleted from the system.

To delete a local user, do the following:

```
$computername = [system.net.dns]::GetHostName()
Function delete-user { param($Computer, $username)
    $ADSI = [ADSI]"WinNT://$Computer"
    $ADSI.Delete("user", "$username")
}
delete-user $computername "remLocalAccount"
```

The output of this is shown in the following screenshot:

```
PS C:\> $computername = [system.net.dns]::GetHostName()
PS C:\> Function delete-user { param($Computer, $username)
>> $ADSI = [ADSI]"WinNT://$Computer"
>> $ADSI.Delete("user", "$username")
>>
>> delete-user $computername "remLocalAccount"
>>
PS C:\>
```

This script displays how to create a function to delete local users from a system. You first define the computer name by leveraging the `[system.net.dns]` .NET assembly, using its `GetHostName()` method, and saving the name to the `$computername` variable. You then create a function named `delete-user` with the `param` arguments of `$computer` and `$username`. Next, you create the ADSI connection string into the computer by calling `[ADSI] "WinNT://$computer"` and setting it to the variable `$ADSI`. You then invoke the `$ADSI.Delete()` method with the arguments of `user` and `$username` to delete the object type of `User` with the name of `$username`. You finally complete the script by calling the `delete-user` function with `$computername` and `remLocalAccount`. This deletes the user `remLocalAccount`.

Managing local groups

To create a group leveraging the ADSI connection, you first have to declare the connection by calling `[ADSI] "WinNT://SystemName"` and setting it to the `$ADSI` variable. You then leverage the `.create()` method of the connection variable `$ADSI`. Thereafter, you define a new variable for the group, such as `$group`, and set it equal to the object output of the `.create()` method with the arguments of `("Group", "Group Name")`.

Finally, you can commit the group to the SAM by declaring \$group with the .setinfo() method. After creating the security account, you can add a description for the group by using the \$group.description method and setting it equal to the description you desire. You would then have to call the \$group.setinfo() method to commit the change to the SAM.

To create local groups, do the following:

```
Function create-group{ param($computer, $groupname, $description)
    $ADSI = [ADSI]"WinNT://$Computer"
    $Group = $ADSI.Create("Group", $groupname)
    $Group.setinfo()
    $Group.Description = "$description"
    $Group.setinfo()
}
# Create the MyLocalGroup
create-group $computername "MyLocalGroup" "This is a test local group"
# Create the remLocalGroup
create-group $computername "remLocalGroup" "This is a test local group"
```

The output of this is shown in the following screenshot:

```
PS C:\> Function create-group{ param($computer, $groupname, $description)
>> $ADSI = [ADSI]"WinNT://$Computer"
>> $Group = $ADSI.Create("Group", $groupname)
>> $Group.setinfo()
>> $Group.Description = "$description"
>> $Group.setinfo()
>>
>> # Create the MyLocalGroup
>> create-group $computername "MyLocalGroup" "This is a test local group"
>> # Create the remLocalGroup
>> create-group $computername "remLocalGroup" "This is a test local group"
>>
PS C:\>
```

This script displays how to create functions to create local groups on a system. You start by defining the computer name by leveraging the [system.net.dns] .NET assembly, using its GetHostName() method, and saving the name to the \$computername variable. You then create a function named create-group with the param arguments of \$computer,

\$groupname, and \$description. Next, you create the ADSI connection string into the computer by calling [ADSI]"WinNT://\$computer" and setting it to the variable \$ADSI. Following this, you invoke the \$ADSI.Create() method with the arguments of Group and \$groupname to create the object type of Group with the name of \$groupname, and set it to the \$group variable. You then call the \$Group.setinfo() method to set the information into the SAM. Finally, the function creates a description by calling the \$Group.description method and setting it equal to the \$description variable.

You commit the description changes by calling the \$Group.setinfo() method to set the description information into the SAM. The script is completed by calling the create-group function with the \$computername variable, the MyLocalGroup group name, and the This is a test local group description. This creates a new group named MyLocalGroup with the description of This is a test local group. You then call the create-group function again with the \$computername variable, the remLocalGroup group name, and the This is a test local group description. This creates a new group named remLocalGroup with the description of This is a test local group.

To delete a group, you leverage a similar methodology as creating a group. You first have to declare the connection by calling [ADSI]"WinNT://SystemName" and setting it to the \$ADSI variable. You then leverage the .Delete() method of the connection variable \$ADSI and reference the arguments of ("Group", "Group Name"). After running the \$ADSI.Delete("Group", "Group Name") method, the group will be deleted from the system.

To delete a local group, do the following:

```
$computername = [system.net.dns]::GetHostName()
Function delete-group { param($computer, $groupname)
    $ADSI = [ADSI]"WinNT://$Computer"
    $ADSI.Delete("Group", $groupname)
}
# Delete the local group remLocalGroup
delete-group $computername "remLocalGroup"
```

The output of this is shown in the following screenshot:

```
PS C:\> $computername = [system.net.dns]::GetHostName()
PS C:\> Function delete-group { param($computer, $groupname)
>> $ADSI = [ADSI]"WinNT://$Computer"
>> $ADSI.Delete("Group", $groupname)
>>
>> # Delete the local group remLocalGroup
>> delete-group $computername "remLocalGroup"
>>
PS C:\>
```

This script displays how to create functions to remove local groups from a system. You start by defining the computer name by leveraging the `[system.net.dns]` .NET assembly, using its `GetHostName()` method, and saving the name to the `$computername` variable. You then create a function named `delete-group` with the `param` arguments of `$computer` and `$groupname`. You then create the ADSI connection string into the computer by calling `[ADSI]"WinNT://$computer"` and setting it to the variable `$ADSI`. Next, you invoke the `$ADSI.Delete()` method with the arguments of `Group` and `$Groupname` to delete the object type of `User` with the name of `$username`. Finally, you call the `delete-group` function with `$computername` and `remLocalGroup`. This deletes the group `remLocalGroup`. After execution of the whole script, you will have the `MyLocalGroup` group created on the system.

There may be instances where you want to add users to a local group. To add members to a group, you first have to define where the user object exists on the system using ADSI, and store the user object in a variable. This is done through defining a variable and setting it equal to an ADSI path like `[ADSI]"WinNT://SystemName/User"`. You then need to define a variable for the destination group object using an ADSI path. This is done through defining a variable and setting it equal to an ADSI path like `[ADSI]"WinNT://SystemName/MyGroup, Group"`. The `, Group` argument tells the ADSI provider that you aren't looking for a user named `MyGroup`, but rather for a group named `MyGroup`.

The last step to adding a user to a group is to leverage the group object method of `.Add()`. As an argument to the method, you specify the user object you want to add to that group. After executing this, the user will

be added to the group.

To create a function to add a user to a group, do the following:

```
$computername = [system.net.dns]::GetHostName()
Function add-groupmember { param($computer, $user, $groupname)
    $userADSI = ([ADSI]"WinNT://$computer/$user").path
    $group = [ADSI]"WinNT://$computer/$groupname, group"
    $group.Add($userADSI)
}
add-groupmember $computername "svcLocalAccount" "MyLocalGroup"
add-groupmember $computername "svcLocalAccount"
"Administrators"
```

The output of this is shown in the following screenshot:

```
PS C:\> $computername = [system.net.dns]::GetHostName()
PS C:\> Function add-groupmember { param($computer, $user, $groupname)
>> $userADSI = ([ADSI]"WinNT://$computer/$user").path
>> $group = [ADSI]"WinNT://$computer/$groupname, group"
>> $group.Add($userADSI)
>>
>> add-groupmember $computername "svcLocalAccount" "MyLocalGroup"
>> add-groupmember $computername "svcLocalAccount" "Administrators"
>>
PS C:\>
```

This script displays how to create functions to add local users to local groups. You start by defining the computer name by leveraging the `[system.net.dns]` .NET assembly, using its `GetHostName()` method, and saving the name to the `$computername` variable. You then create a function named `add-groupmember` with the `param` arguments of `$computer`, `$user`, and `$groupname`. After this, you retrieve the ADSI connection string for the user by calling

`([ADSI]"WinNT://$computer/$user").path` and setting it to the variable `$userADSI`. This differs from the previous examples, where it doesn't actually make the connection to the ADSI adapter, it is just a user object location reference in ADSI format.

You continue by retrieving the ADSI connection string into the group on the computer, by calling `[ADSI]"WinNT://$computer,$groupname, group"` and setting it to the variable `$group`. You then invoke the `add` method of the `$userADSI` user object in the group defined in `$group` by

calling `$group.Add($userADSI)`. After execution, the group defined in `$group` will contain the user referenced in `$userADSI`.

Finally, you call the `add-groupmember` function with the arguments of `$computername`, `svcLocalAccount`, and `MyLocalGroup`. This will add `svcLocalAccount` to the `MyLocalGroup` local group. You also call the `add-groupmember` function with the arguments of `$computername`, `svcLocalAccount`, and `Administrators`. This will add `svcLocalAccount` to the `Administrators` local group.

Removing a user from a group is very similar to adding a user to a group. To remove members from a group, you first have to define where the user object exists on the system using ADSI and store the user object in a variable. This is done through defining a variable and setting it equal to an ADSI path like `[ADSI]"WinNT://SystemName/User"`. You then need to define a variable for the destination group object using an ADSI path. This is done through defining a variable and setting it equal to an ADSI path like `[ADSI]"WinNT://SystemName/MyGroup, Group"`. The final step to removing a user from a group is by leveraging the `group` object method of `.Remove()`. As an argument to the method, you specify the user object you want to remove from that group. After executing this, the user will be removed from the group.

To create a function to remove a user from a group, do the following:

```
$computername = [system.net.dns]::GetHostName()
Function delete-groupmember { param($computer, $user,
$groupname)
    $userADSI= ([ADSI]"WinNT://$computer/$user").path
    $group = [ADSI]"WinNT://$computer/$groupname, group"
    $group.Remove($userADSI)
}
delete-groupmember $computername "svcLocalAccount"
"Administrators"
```

The output of this is shown in the following screenshot:

```
PS C:\> $computername = [system.net.dns]::GetHostName()
PS C:\> Function delete-groupmember { param($computer, $user, $groupname)
>> $userADSI= ([ADSI]"WinNT://$computer/$user").path
>> $group = [ADSI]"WinNT://$computer/$groupname, group"
>> $group.Remove($userADSI)
>> }
>> delete-groupmember $computername "svcLocalAccount" "Administrators"
>>
PS C:\>
```

This script displays how to create functions to add local users to local groups. You start by defining the computer name by leveraging the `[system.net.dns]` .NET assembly, using its `GetHostName()` method, and saving the name to the `$computername` variable. You then create a function named `delete-groupmember` with the `param` arguments of `$computer`, `$user`, and `$groupname`. After that, you retrieve the ADSI connection string for the user by calling

`([ADSI]"WinNT://$computer/$user").path` and setting it to the variable `$userADSI`. You continue by retrieving the ADSI connection string into the group on the computer by calling

`[ADSI]"WinNT://$computer,$groupname, group,"` and setting it to the variable `$group`. You then invoke the `Remove` method of the `$userADSI` user object in the group defined in `$group` by calling

`$group.Remove($userADSI)`. After execution, the group defined in `$group` will not contain the user referenced in `$userADSI`. Finally, you call the `delete-groupmember` function with the arguments of `$computername`, `svcLocalAccount`, and `Administrators`. This will delete `svcLocalAccount` from the `Administrators` local group.

Querying for local users and groups

As you are creating users and groups, you may want to verify that a user or group exists prior to creation or deletion. To verify that a user exists, you can leverage the `::Exists` argument to the `[ADSI]` adapter. To do this, you call `[ADSI]::Exists` followed by the ADSI path of the user that you want to modify, such as `WinNT://ComputerName/Username`, and the reference of the `User` object type to complete the ADSI call. This tells the ADSI provider to search `Computername` specified for the object type of `User` for the user named `Username`.

If you want to verify that a group exists, you can leverage the `::Exists` argument to the [ADSI] adapter. To do this, you declare `[ADSI]::Exists`, followed by the ADSI location of the user that you want to modify, such as `WinNT://ComputerName/GroupName`, and the reference of `Group` object type to complete the ADSI call. This tells the ADSI provider to search `Computername` specified for the object type of `Group` for the group named `GroupName`.

To search a computer for user and a group object, do the following:

```
$computername = [system.net.dns]::GetHostName()
Function get-ADSIsearch { param($computer, $objecttype,
$object)
    $test = [ADSI]::Exists("WinNT://$computer/$object,
$objecttype")
    If ($test) { Write-host "Local $objecttype Exists. Test
Variable Returned: $test" }
    If (!$test) { Write-host "Local $objecttype Does Not Exist.
Test Variable Returned: $test" }
}
get-ADSIsearch $computername "User" "svcLocalAccount"
get-ADSIsearch $computername "Group" "MyLocalGroup"
get-ADSIsearch $computername "Group" "NotARealGroup"
```

The output of this is shown in the following screenshot:

```
PS C:\> $computername = [system.net.dns]::GetHostName()
PS C:\> Function get-ADSIsearch { param($computer, $objecttype, $object)
>> $test = [ADSI]::Exists("WinNT://$computer/$object, $objecttype")
>> If ($test) { Write-host "Local $objecttype Exists. Test Variable Returned: $test" }
>> If (!$test) { Write-host "Local $objecttype Does Not Exist. Test Variable Returned: $test" }
>>
>> get-ADSIsearch $computername "User" "svcLocalAccount"
>> get-ADSIsearch $computername "Group" "MyLocalGroup"
>> get-ADSIsearch $computername "Group" "NotARealGroup"
>>
Local User Exists. Test Variable Returned: True
Local Group Exists. Test Variable Returned: True
Local Group Does Not Exist. Test Variable Returned: False
PS C:\> _
```

This example displays how to create a function query for a system to see if a user or group has already been created on a system. You start by defining the computer name by leveraging the `[system.net.dns]` .NET assembly, using its `GetHostName()` method, and saving the name to the `$computername` variable. You then create a function named `get-`

ADSI Search with the param arguments of \$computer, \$objecttype, and \$object. Then, you leverage the ADSI adapter with the ::Exists method by calling [ADSI]::Exists("WinNT://\$computer/\$object, \$object type") and setting it to the variable \$test. Next, you create an implied True IF statement of (\$test) where if True, it will use the write-host cmdlet to print Local \$objecttype Exists. Test Variable Returned: \$test. You then create an implied False IF statement of (!\$test) where if False, it will use the write-host cmdlet to print Local \$objecttype Does Not Exist. Test Variable Returned: \$test.

Finally, you test three objects by first calling get-ADSI Search function with the arguments of \$computername, User, and svcLocalAccount. This will return True because the svcLocalAccount user is on the system. The script will print to the screen Local User Exists. Test Variable Returned: True. Second, you call the get-ADSI Search function with the arguments of \$computername, Group, and MyLocalGroup. This will return True because the MyLocalGroup group exists on the system. The script will print to the screen Local Group Exists. Test Variable Returned: True. The third call is to the get-ADSI Search function with the arguments of \$computername, Group, and NotARealGroup. This will return False because the NotARealGroup group does not exist on the system. The script will print to the screen Local Group Does Not Exist. Test Variable Returned: False.

If you want to view the members of a group, you can leverage an ADSI provider to get the group members. First, you need to define the group you want to query. This is done through defining a group variable and setting it equal to an ADSI path like

[ADSI]"WinNT://SystemName/Groupname". You then need to query the members of the group and search through the individual members. This is done through creating an array of the ADSI group object members by invoking the Members method. In code, this looks like
@(\$groupvariable.Invoke("Members")).

After you receive the ADSI group object members, you need to retrieve the Name property of the individual group members. You pipe the results

of the array to a `foreach` loop and get the name property by typing | `foreach { $_.GetType().InvokeMember("Name", 'GetProperty', $null, $_, $null) }`. After taking the pipeline and getting the `Name` property from each object, a list of group members' names is printed to the console.

To get the list of the members of a local group, do the following:

```
$computername = [system.net.dns]::GetHostName()
Function get-groupmember { param($computer,$groupname)
    $group = [ADSI]"WinNT://$computer/$groupname"
    @{$group.Invoke("Members"))} | foreach {
        $_.GetType().InvokeMember("Name", 'GetProperty', $null, $_,
        $null) }

}
$members = get-groupmember $computername "MyLocalGroup"
Write-host "The members for MyLocalGroup are: $members"
$members = get-groupmember $computername "Administrators"
Write-host "The members for Administrators are: $members"
```

The output of this is shown in the following screenshot:

```
PS C:\> $computername = [system.net.dns]::GetHostName()
PS C:\> Function get-groupmember { param($computer,$groupname)
>> $group = [ADSI]"WinNT://$computer/$groupname"
>> @{$group.Invoke("Members"))} | foreach { $_.GetType().InvokeMember("Name", 'GetProperty', $null, $_, $null) }
>>
>> }
>> $members = get-groupmember $computername "MyLocalGroup"
>> Write-host "The members for MyLocalGroup are: $members"
>> $members = get-groupmember $computername "Administrators"
>> Write-host "The members for Administrators are: $members"
>>
The members for MyLocalGroup are: svcLocalAccount
The members for Administrators are: Administrator Brenton
```

This example displays how to query the group members of `MyLocalGroup` and `Administrators` group on a computer. You start by defining the computer name by leveraging the `[system.net.dns]` .NET assembly, using its `GetHostName()` method, and saving the name to the `$computername` variable. After that, you create a function named `get-groupmember` with the `param` arguments of `$computer` and `$groupname`. You then retrieve the ADSI connection string for the group by calling `[ADSI]"WinNT://$computer/$groupname"` and setting it to the variable `$group`.

You then create an array of members by typing
@(\$group.Invoke("Members")). Next, you pipe those results to a foreach loop and get the Name property for each of the results by typing
{\$_.GetType().InvokeMember("Name", 'GetProperty', \$null, \$_, \$null)}. This will return the results of the individual member's names within the function. You then close the get-groupmember function.

After defining the function, you call the get-groupmember function for the MyLocalGroup group and set the results to the \$members variable. You then print to the screen The members for MyLocalGroup are: \$members. The output to the console is The members for MyLocalGroup are:
svcLocalAccount.

Finally, you call the get-groupmember function for the Administrators group and set the results to the \$members variable. You then print to the screen The members for Administrators are: \$members. The output to the console is The members for Administrators are: Administrator Brenton.

Note

For detailed information on GetType() and InvokeMember(), you can refer to [https://msdn.microsoft.com/en-us/library/de3dhzwy\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/de3dhzwy(v=vs.110).aspx).

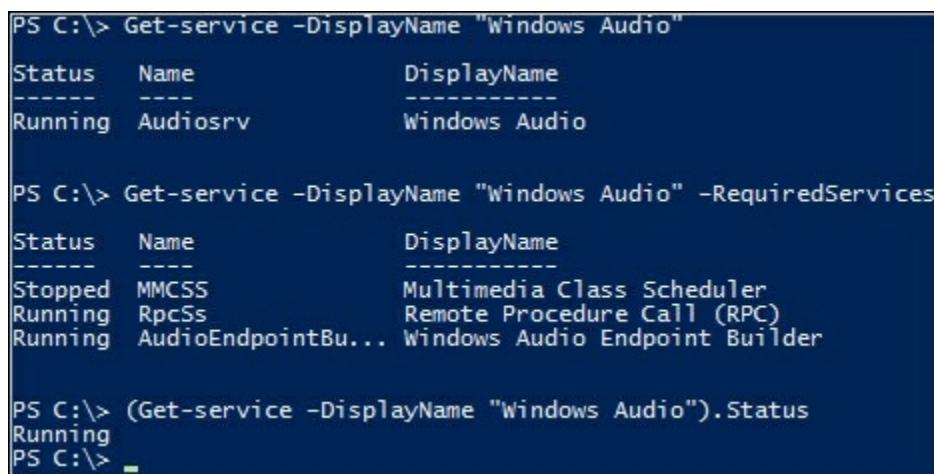
Managing Windows services

When you are working with Microsoft-based systems, there may be times where you need to interact with Windows services. PowerShell offers a variety of cmdlets that enable you to work with these services. To start, you can review the services on a system by leveraging the `get-service` cmdlet. By calling the `get-service` cmdlet, you can retrieve the full list of services on a system. If you want to obtain a filtered view into a specific service, you can leverage the `-Name` parameter, referencing a specific name of a service. After executing this command, you will see `Status`, `Name`, and `DisplayName` of the service. You may also issue the `-RequiredServices` parameter to display the services that are required to be running, for that particular service to be functional. You may also use `-DependentServices` to view the services that are dependent on that service.

To use the `get-service` cmdlet to query the Windows Audio Service, do the following:

```
Get-service -DisplayName "Windows Audio"
Get-service -DisplayName "Windows Audio" -RequiredServices
(Get-service -DisplayName "Windows Audio").Status
```

The output of this is shown in the following screenshot:



The screenshot shows a PowerShell window with three commands entered and their outputs:

```
PS C:\> Get-service -DisplayName "Windows Audio"
Status     Name           DisplayName
----     ----           -----
Running   Audiosrv      Windows Audio

PS C:\> Get-service -DisplayName "Windows Audio" -RequiredServices
Status     Name           DisplayName
----     ----           -----
Stopped   MMCSS         Multimedia Class Scheduler
Running   RpcSs          Remote Procedure Call (RPC)
Running   AudioEndpointBu... Windows Audio Endpoint Builder

PS C:\> (Get-service -DisplayName "Windows Audio").Status
Running
PS C:\>
```

This example displays how to get information about the Windows Audio

Service. You start by calling the `get-service` cmdlet, leveraging the `-DisplayName` parameter, and referencing the Windows Audio Windows service. After executing, you see the Status, Name, and DisplayName fields printed to the PowerShell window. You then use the `get-service` cmdlet with the `-DisplayName` parameter referencing Windows Audio, and the `-RequiredServices` parameter. After executing the `get-service` command, you will see Status, Name, and DisplayName of all the services that are required for the Windows Audio Windows service to function properly. The last call you make leverages the `get-service` cmdlet with the `-DisplayName` parameter referencing the Windows Audio. This whole statement is wrapped in parentheses followed by the dot notation of `.Status`. This returns the current status of the Windows Audio Windows service, which is `Running`.

In instances where you want to start, restart, and stop services, you may leverage the `start-service`, `restart-service`, and `stop-service` cmdlets. To start a service, you can call the `start-service` cmdlet, followed by the `-Name` or `-DisplayName` parameters with the corresponding service name. After execution, the service will change status from `Stopped` to `StartPending`, and when it has successfully started, it will change its status to `Running`.

To stop a service, you can call the `stop-service` cmdlet, followed by the `-Name` or `-DisplayName` parameters with the corresponding service name. After execution, the service will change status from `Running` to `StopPending`, and when it has successfully stopped, it will change the status to `Stopped`.

To restart a service, you can call the `restart-service` cmdlet, followed by the `-Name` or `-DisplayName` parameters with the corresponding service name. After execution, the service will change status from `Running` to `StopPending`, `StopPending` to `Stopped`, `Stopped` to `StartPending`, and when it has successfully restarted, it will change the status to `Running`.

To stop and start the Windows Audio service, do the following:

```
stop-service -DisplayName "Windows Audio"
(Get-service -DisplayName "Windows Audio").Status
start-service -DisplayName "Windows Audio"
(Get-service -DisplayName "Windows Audio").Status
```

The output of this is shown in the following screenshot:

```
PS C:\> stop-service -DisplayName "Windows Audio"
PS C:\> (Get-service -DisplayName "Windows Audio").Status
Stopped
PS C:\> start-service -DisplayName "Windows Audio"
PS C:\> (Get-service -DisplayName "Windows Audio").Status
Running
PS C:\>
```

This example shows how to start and stop Windows services. You first stop the Windows Audio service by leveraging the `stop-service` cmdlet, with the `-DisplayName` parameter referencing the `Windows Audio` display name. You then get the current `Status` of the service by executing `get-service`, with the `-DisplayName` parameter referencing the `Windows Audio` display name. You encapsulate that in parentheses and leverage the dot notation of `.Status` to print the current status to the screen. The console will return the status of `Stopped`.

You then start the Windows Audio service by leveraging the `start-service` cmdlet, with the `-DisplayName` parameter referencing the `Windows Audio` display name. You then get the current `Status` of the service by executing `get-service`, with the `-DisplayName` parameter referencing the `Windows Audio` display name. You encapsulate that in parentheses and leverage the dot notation of `.Status` to print the current status to the screen. The console will return the status of `Running`.

You also have the ability to modify different aspects of the Windows services by using the `set-service` cmdlet. The `set-service` cmdlet can modify the service descriptions, start-up types, and even the display names for services. Since Windows does not allow you to modify running services, you first have to leverage the `stop-service` cmdlet to stop the service for editing.

If you want to modify the start-up type for a service, you can leverage

the `set-service` cmdlet with `-Name` parameter with the corresponding service name. You then include the `-StartupType` parameter with `Automatic` for automatic start-up, `Manual` for manual start-up, or `Disabled` to disable the service start-up. To view the changes to the service start-up type, you will need to leverage the `get-wmiobject` cmdlet referencing the `win32_service` class, and the `-filter` parameter referencing the `DisplayName='Display Name'`.

Tip

While in most cases, you can leverage the `get-service` cmdlet to display the properties of a service, there are certain properties that are not made available to the cmdlet. As a result, you may have to directly query WMI using `get-wmiobject` to view all of the properties for that service. To view all of the properties available to `get-service` or through WMI, you can pipe `|` the results to `get-member`, and it will display all available properties that you can view.

To change the Windows Audio `-StartupType` parameter, do the following:

```
(get-wmiobject win32_service -filter "DisplayName='Windows
Audio'").StartMode
stop-service -name "Audiosrv"
set-service -name "Audiosrv" -startup "Manual"
(get-wmiobject win32_service -filter "DisplayName='Windows
Audio'").StartMode
set-service -name "Audiosrv" -startup "Automatic"
(get-wmiobject win32_service -filter "DisplayName='Windows
Audio'" ).StartMode
Start-service -name "Audiosrv"
```

The output of this is shown in the following screenshot:

```
PS C:\> (get-wmiobject win32_service -filter "DisplayName='Windows Audio'").StartMode
Auto
PS C:\> stop-service -name "Audiosrv"
PS C:\> set-service -name "Audiosrv" -startup "Manual"
PS C:\> (get-wmiobject win32_service -filter "DisplayName='Windows Audio'").StartMode
Manual
PS C:\> set-service -name "Audiosrv" -startup "Automatic"
PS C:\> (get-wmiobject win32_service -filter "DisplayName='Windows Audio'" ).StartMode
Auto
PS C:\> Start-service -name "Audiosrv"
PS C:\> _
```

This example shows how to change the start-up of a Window service on a system. You start by querying the system to see what the existing StartMode is. To do this, you have to leverage the `get-wmiobject` cmdlet referencing the `win32_service` class. You leverage the `-filter` parameter with the filter options of `DisplayName='Windows Audio'`. You then encapsulate that statement in parenthesis and leverage the dot notation of `.StartMode` to print to the screen the start mode of a system. The command will print `Auto` to the screen, designating that the service start-up type is set to `Automatic`. Thereafter, you stop the service by calling the `stop-service` cmdlet with the `-name` parameter referencing the `Audiosrv` service name. You then configure the service start-up type to be `Manual` by calling the `set-service` cmdlet with the `-name` parameter referencing `AudioSrv`, and the `-startup` parameter referencing `Manual`. After setting this command, you verify the start-up change by using the `get-wmiobject` cmdlet referencing the `win32_service` class. You leverage the `-filter` parameter with the filter options of `DisplayName='Windows Audio'`. Next, you encapsulate that statement in parenthesis and leverage the dot notation of `.StartMode` to print to the screen the start mode of a system. The command will print `Manual` to the screen designating that the service start-up type is set to `Manual`.

After this, you set the service back to `Automatic` by calling the `set-service` cmdlet with the `-name` parameter referencing `AudioSrv`, and the `-startup` parameter referencing `Automatic`. After setting this command, you verify the start-up change by using the `get-wmiobject` cmdlet referencing the `win32_service` class. You leverage the `-filter` parameter with the filter options of `DisplayName='Windows Audio'`. You then encapsulate that statement in parenthesis and leverage the dot notation of `.StartMode` to print to the screen the start mode of a system. The command will print `Auto` to the screen designating that the service start-up type is set to `Automatic`. After final configuration, you start the service by calling the `start-service` cmdlet, with the `-name` parameter referencing the `Audiosrv` service name.

If you want to modify a service's description, you first need to stop the service leveraging the `stop-service` cmdlet. You then call the `set-service` cmdlet with the `-Name` name of the service, and the `-Description` parameter with the description that you want set for the particular service. The `Description` property is unique as it is not made available to the `get-service` cmdlet. To get around this, you need to leverage the `get-wmiobject` cmdlet. To view the description, you can use the `get-wmiobject` cmdlet referencing the `win32_service` class, with the `-filter` parameter referencing `DisplayName='Display Name'`. After executing this script, the description will print to the screen. After setting the description, you can start the service by using the `start-service` cmdlet.

To set the description for the Windows Audio service, do the following:

```
$olddesc = (get-wmiobject win32_service -filter "DisplayName='Windows Audio'").description
stop-service -DisplayName "Windows Audio"
Set-service -name "Audiosrv" -Description "My New Windows Audio Description."
(get-wmiobject win32_service -filter "DisplayName='Windows Audio'").description
Set-service -name "Audiosrv" -Description $olddesc
(get-wmiobject win32_service -filter "DisplayName='Windows Audio'").description
start-service -DisplayName "Windows Audio"
```

The output of this is shown in the following screenshot:



```
PS C:\> $olddesc = (get-wmiobject win32_service -filter "DisplayName='Windows Audio'").description
PS C:\> stop-service -DisplayName "Windows Audio"
PS C:\> Set-service -name "Audiosrv" -Description "My New Windows Audio Description."
PS C:\> (get-wmiobject win32_service -filter "DisplayName='Windows Audio'").description
My New Windows Audio Description.
PS C:\> Set-service -name "Audiosrv" -Description $olddesc
PS C:\> (get-wmiobject win32_service -filter "DisplayName='Windows Audio'").description
Manages audio for Windows-based programs. If this service is stopped, audio devices and effects will not function properly. If this service is disabled, any services that explicitly depend on it will fail to start
PS C:\> start-service -DisplayName "Windows Audio"
PS C:\> _
```

This example displays how to change the description for a Windows service, and set it back to the original description. You start by querying the system to see what the existing description is. To do this, you have to

leverage the `get-wmiobject` cmdlet referencing the `win32_service` class. You leverage the `-filter` parameter with the filter options of `DisplayName='Windows Audio'`. You then encapsulate that statement in parenthesis and leverage the dot notation of `.description`. The output, which is the Windows service description, is then set to the variable `$olddesc`. You then stop the service with the `stop-service` cmdlet, and the `-DisplayName` parameter referencing Windows Audio. To set the description, you use the `set-service` cmdlet with the `-name` parameter set to `Audiosrv`, and the `-description` parameter set to `My New Windows Audio Description`.

After setting the description, you query the system with the `get-wmiobject` cmdlet referencing the `win32_service` class, and the `-filter` parameter with the filter options of `DisplayName='Windows Audio'`. You then encapsulate that statement in parenthesis and leverage the dot notation of `.description`. The output from this will be the current description, which is `My New Windows Audio Description`. To set the description back to the original, you use the `set-service` cmdlet with the `-name` parameter set to `Audiosrv`, and the `-description` parameter referencing the `$olddesc` variable. After setting the description back to the original description, you query the system with the `get-wmiobject` cmdlet referencing the `win32_service` class, and the `-filter` parameter with the filter options of `DisplayName='Windows Audio'`. You then encapsulate that statement in parenthesis and leverage the dot notation of `.description`. The output from this will be the current description, which is `Manages audio for Windows-based programs`. If this service is stopped, audio devices and effects will not function properly. If this service is disabled, any services that explicitly depend on it will fail to start. You complete this process by starting the Windows Audio service. You use the `start-service` cmdlet with the `-DisplayName` parameter referencing `Windows Audio`.

Managing Windows processes

There may be times when, during scripting, you need to check if there is a running process on a system. PowerShell offers the `get-process` cmdlet to search for available processes on a system. By running the `get-process` cmdlet alone, you will get a report of all the running services on the system. The default record set that is returned about the running services include:

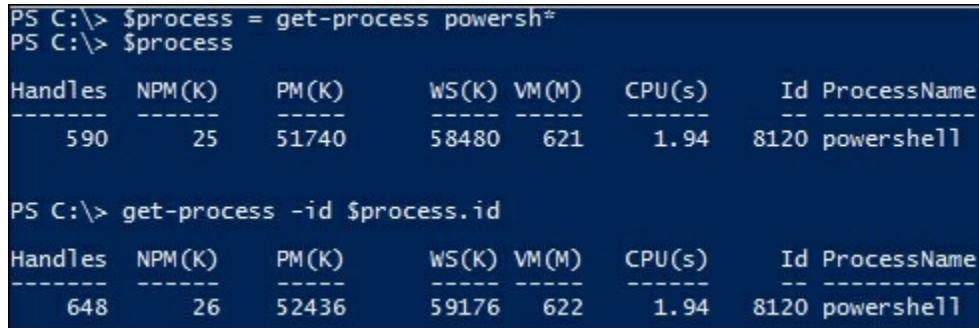
- **Handles:** The number of thread handles that are being used by a particular process
- **NPM (K):** Non Paged Memory is the memory that is solely in physical memory, and not allocated to the page file that is being used by a process
- **PM (K):** Pageable Memory is the memory that is being allocated to the page file that is used by a process
- **WS(K):** Working Set is the memory recently referenced by a process
- **VM(M):** Virtual Memory is the amount of virtual memory that is being used by a process
- **CPU(s):** Processor time, or the time the CPU is utilizing a process
- **ID:** An assigned Unique ID to a Process
- **Process name:** The name of the process in memory

Typically, when you query the active running processes on a system, you will be looking for a particular process. To do this, you can leverage the `get-process` cmdlet with the `-name` parameter referencing `process name` to view information about that particular process. You can also leverage the asterisk (*) to be a wild card to query processes that are like the partial word you specify. You may also directly reference the Process ID of a process if you invoke the `-ID` parameter with an ID of a process. If you want more information about the process that is running, you can also leverage the `-fileversioninfo` parameter to pull `ProductVersion`, `FileVersion`, and `Filename` information from the process. In instances where you need to find all of the modules, or DLL references that are loaded by a process, you may also leverage the `-module` parameter.

To search for a process by using a wild card and get a process by a process ID, do the following:

```
$process = get-process powersh*
$process
get-process -id $process.id
```

The output of this is shown in the following screenshot:



PS C:\> \$process = get-process powersh*
PS C:\> \$process
Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName
----- -- -- -- -- -- --
 590 25 51740 58480 621 1.94 8120 powershell

PS C:\> get-process -id \$process.id
Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName
----- -- -- -- -- -- --
 648 26 52436 59176 622 1.94 8120 powershell

This script displays how to search for a process by a wildcard and obtain the process ID. You also view that same service by calling the process ID of that service. You start by using the `get-process` cmdlet with the `powersh*` searching wild card. The system returns the PowerShell process into the `$process` variable. You then call the `$process` variable to view the information about the PowerShell process. Next, you leverage the `get-process` cmdlet with the `-id` parameter pointing to the `$process` variable referencing the dot notation of `.id`. This returns the same PowerShell process information, as `$process.id` that is being referenced is the process ID of the first search result. After executing this script, you will see the Handles, NPM, PM, WS, VM, CPU, ID, and ProcessName information for the PowerShell process.

To get a process using a wildcard and get its `FileVersionInfo` information, do the following:

```
$process = get-process powersh*
get-process -id $process.id -FileVersionInfo
```

The output of this is shown in the following screenshot:

```
PS C:\> $process = get-process powersh*
PS C:\> get-process -id $process.id -FileVersionInfo
ProductVersion    FileVersion      FileName
-----          -----      -----
6.3.9600.17396   6.3.9600.1739...  C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

This script displays how to search for a process by using a wild card and then use that information to view the file version information. You first start by using the `get-process` cmdlet with the `powersh*` searching wild card. The system returns the PowerShell process into the `$process` variable. You then call the `get-process` cmdlet, and leverage the `-id` parameter pointing to the `$process` variable referencing the dot notation of `.id`. You also call the `-FileVersionInfo` parameter to display the advanced information about the PowerShell process. After executing this script, you will see the `ProductVersion`, `FileVersion`, and `FileName` information about the PowerShell process.

To get a process by a wild card, and get the number of modules for that process, do the following:

```
$process = get-process powersh*
$modules = get-process -id $process.id -module
$modules.count
```

The output of this is shown in the following screenshot:

```
PS C:\> start-process -FilePath notepad.exe
PS C:\> $process = get-process notepad*
PS C:\>
```

This script shows how to search for a process by using a wild card and then use that information to view the module information. You start by using the `get-process` cmdlet with the `powersh*` searching wild card. The system returns the PowerShell process into the `$process` variable. You then call the `get-process` cmdlet, leverage the `-id` parameter pointing to the `$process` variable referencing the dot notation of `.id`, and call the `-module` parameter. You then save the result in a variable

named \$modules. The final step is to count the number of modules that are linked to the PowerShell process by using \$modules.count. After executing this script, you will see that there are 77 module items that make up the PowerShell process.

To start a new process, or invoke a program, you can leverage the `start-process` cmdlet. The proper syntax for using this cmdlet is calling the `start-process` cmdlet and providing the `-filepath` parameter pointing to the location of the item you want to execute. You can then call the optional `-argumentlist` parameter, referencing the parameters that are needed to execute the item, the optional `-verb` parameter to invoke any verbs associated with the file type (such as `Edit`, `Open`, `Play`, `Print`, and `RunAs`), the optional `-NoNewWindow` parameter to not spawn the command in a new PowerShell console, and the optional `-wait` parameter to wait for the process to complete before continuing with the script. If you do not execute the `-wait` parameter, the script will continue to the next step without waiting for the current step to be successful. After starting a process, the process will receive a process ID for which you can reference with the other process cmdlets.

To start a new notepad process, do the following:

```
start-process -FilePath notepad.exe
$process = get-process notepad*
```

The output of this is shown in the following screenshot:

```
PS C:\> start-process -FilePath notepad.exe
PS C:\> $process = get-process notepad*
PS C:\>
```

This script displays how to start a notepad process, and search for the notepad process by using a wild card. You start by using the `start-process` cmdlet with the `-filepath` parameter referencing the `notepad.exe` process. After execution, it will launch `notepad.exe`. You then use the `get-process` cmdlet with the `notepad*` searching wild card. The system returns the PowerShell process object into the `$process`

variable.

To stop a process, or stop a program, you can leverage the `stop-process` cmdlet. The proper syntax for using this cmdlet is calling the `stop-process` cmdlet and providing the `-filepath` parameter pointing to the location of the item you want to terminate. You may also leverage the `-processname` parameter to stop a service by its process name or use wildcards with the `-processname` parameter to end processes that are like the partial word you specify. You can specify the `-id` parameter to terminate a process by its Process ID as well. By default, if you kill a process, it will prompt you for confirmation. The `-force` parameter will force the termination of the process without prompting the user.

To stop the running notepad process, do the following:

```
start-process -FilePath notepad.exe
$process = get-process notepad*
stop-process -ID $process.id
```

The output of this is shown in the following screenshot:

```
PS C:\> start-process -FilePath notepad.exe
PS C:\> $process = get-process notepad*
PS C:\> stop-process -ID $process.id
PS C:\>
```

This script displays how to start a notepad process, search for the notepad process by a wild card, and then use that information to stop the notepad process. You start by using the `start-process` cmdlet with the `-filepath` parameter referencing the `notepad.exe` process. After execution, it will launch `notepad.exe`. You then use the `get-process` cmdlet with the `notepad*` searching wild card. The system returns the PowerShell process into the `$process` variable. Next, you call the `stop-process` cmdlet, and leverage the `-id` parameter pointing to the `$process` variable referencing the dot notation of `.id`. After executing this script, you will see the notepad open and close.

Installing Windows features and roles

Windows Server 2012 SP2, and PowerShell 4.0 introduced new cmdlets to install Windows features through the use of scripts. This provides a further layer of automation to the PowerShell toolset as you can dynamically and completely build servers with a single script.

Tip

If you want to manage the Windows features cmdlets from a Windows 8.1 system, you will first need to install Remote Server Administration Tools. Then you will have to enable Server Manager the feature. This will enable you to manage server-based operating systems such as Windows Server 2012 R2. These can be found at

<http://www.microsoft.com/en-us/download/details.aspx?id=39296>.

To view the features that are available for installation and uninstallation through the cmdlets, you can leverage the `get-windowsfeature` cmdlet. When you call the `get-windowsfeature` cmdlet without parameters, you will find that there are over 260 items that can be individually installed. Each feature on the system is broken up into `Display Names`, `Names`, and `Install State`. These features also have sub-features that can also be installed, uninstalled, or viewed. If you want to dig deeper into a specific feature, you can leverage the `-name` parameter which will pull up the specific information for that particular service.

To view all the Windows feature information that match the word Telnet, do the following:

```
$featureinfo = get-WindowsFeature | Where {$_.DisplayName -match "Telnet"}  
foreach ($feature in $featureinfo) {  
    Write-host "Feature Display Name:" $feature.DisplayName  
    Write-host "Feature Name:" $feature.Name  
    Write-host "Feature Install State:" $feature.InstallState  
    Write-host ""
```

}

The output of this is shown in the following screenshot:

```
PS C:\> $featureinfo = Get-WindowsFeature | `where {$_.Displayname -match "Telnet"}  
PS C:\> foreach ($feature in $featureinfo) {  
    >>     Write-Host "Feature Display Name:" $feature.DisplayName  
    >>     Write-Host "Feature Name:" $feature.Name  
    >>     Write-Host "Feature Install State:" $feature.InstallState  
    >>     Write-Host ""  
    >>  
Feature Display Name: Telnet Client  
Feature Name: Telnet-Client  
Feature Install State: Available  
  
Feature Display Name: Telnet Server  
Feature Name: Telnet-Server  
Feature Install State: Available
```

This example displays how to query properties about services on a system. You start by getting the Windows features that match the word Telnet. You leverage the `get-WindowsFeature` cmdlet, pipe `|` the results to the statement `where {$_.Displayname -match "Telnet"}`. You place the output of that command into the `$featureinfo` variable and then create a `foreach` loop to query each `$feature` in `$featureinfo`. You then call the `write-host` cmdlet and print to screen `Feature DisplayName:` with the `$feature` variable referencing the dot notation of `.Displayname`. Thereafter, you use the `write-host` cmdlet, print to screen `Feature Name:` with the `$feature` variable referencing the dot notation of `.name`.

Finally, you use the `write-host` cmdlet, print to screen `Feature Install State:` with the `$feature` variable referencing the dot notation of `.InstallState`. After executing this command, the console will print to screen Feature Display Name, Feature Name, and Feature Install State for both Telnet-Client and Telnet-Server.

If you want to install a feature, you can use the `install-windowsfeature` cmdlet, with the `-name` parameter referencing the feature you want to install. You can also designate the `-InstallAllSubFeature` parameter, which installs all of the sub-features for the feature you are installing on the system. You can designate the `-`

`IncludeManagementTools` parameter to include the management tools for the specific feature you are installing. It also allows for a configuration file with `-ConfigurationFilePath` for advanced configuration options for individual feature installations. The `install-windowsfeature` cmdlet also supports offline editing of VHD's features. If you specify the `-vhd` parameter pointing to a VHD location, you can add a feature in an offline servicing mode.

To install windows features, do the following:

```
Install-WindowsFeature -name Telnet-Client -  
IncludeAllSubFeature -IncludeManagementTools  
install-WindowsFeature -name Telnet-Server -  
IncludeAllSubFeature -IncludeManagementTools
```

The output of this is shown in the following screenshot:

```
PS C:\> install-WindowsFeature -Name "Telnet-Server" -IncludeAllSubFeature -IncludeManagementTools  
Success Restart Needed Exit Code      Feature Result  
----- ----- -----  
True    No        Success      {Telnet Server}  
WARNING: Windows automatic updating is not enabled. To ensure that your newly-installed role or feature is  
automatically updated, turn on Windows Update.  
  
PS C:\> install-WindowsFeature -Name "Telnet-Client" -IncludeAllSubFeature -IncludeManagementTools  
Success Restart Needed Exit Code      Feature Result  
----- ----- -----  
True    No        NoChangeNeeded {}
```

This example displays how to install the `Telnet-Server` and `Telnet-Client` Windows features, and their management tools, on a system. You start by leveraging the `install-windowsfeature` cmdlet with the `-name` parameter pointing to `Telnet-Server`. You then use the `-IncludeAllSubfeature` parameter to install all the sub-features, and the `-IncludeManagementTools` parameter to install all the management tools. You proceed to install `Telnet-Client` by using the `install-windowsfeature` cmdlet with the `-name` parameter pointing to `Telnet-Client`. You then use the `-IncludeAllSubFeature` parameter to install all the sub-features, and the `-IncludeManagementTools` parameter to install all the management tools.

After running this script, both Telnet-Server and Telnet-Client, and their sub-features and management tools, will be installed on the system.

To uninstall a Windows feature, you can leverage the `uninstall-windowsfeature` cmdlet with the `-name` parameter to specify a Windows feature you want to remove. You can specify the `-restart` parameter to restart the system after the feature is uninstalled. You may also want to use the `-IncludeManagementTools` parameter to also uninstall the management tools for the feature.

To uninstall windows features, do the following:

```
Uninstall-WindowsFeature -Name "Telnet-Server"  
uninstall-WindowsFeature -Name "Telnet-Client"
```

The output of this is shown in the following screenshot:

The screenshot shows two separate PowerShell command executions. The first command is `PS C:\> Uninstall-WindowsFeature -Name "Telnet-Server"`. The output is a table with four columns: Success, Restart Needed, Exit Code, and Feature Result. The row for Telnet Server has values: True, No, Success, and {Telnet Server}. The second command is `PS C:\> Uninstall-WindowsFeature -Name "Telnet-Client"`. The output is a similar table for the Telnet Client feature, also showing success with no restart needed and a success exit code.

Success	Restart Needed	Exit Code	Feature Result
True	No	Success	{Telnet Server}

Success	Restart Needed	Exit Code	Feature Result
True	No	Success	{Telnet Client}

This example displays how to uninstall the Telnet-Server and Telnet-Client Windows features from a system. You start by using the `uninstall-windowsfeature` cmdlet and using the `-name` parameter pointing to Telnet-Server. After execution, the console prints to the window the Success, Restart Needed, Exit Code, and Feature Result properties. You then use the `uninstall-windowsfeature` cmdlet using the `-name` parameter pointing to Telnet-Client. After execution, the console prints to the window the Success, Restart Needed, Exit Code, and Feature Result properties. After running this code, both the Telnet-Server and Telnet-Client Windows Features are successfully uninstalled from the system.

Summary

This chapter provided a good view of how to manage the basic functions of Microsoft systems. You first learned how to leverage the active directory services interface (ADSI) adapter to make a connection to the local system. You learned how to use the ADSI adapter to create an ADSI adapter variable. You now understand how to leverage the `.Create()` method to create new users and groups and how to use the `.Delete()` method to delete users and groups. You also got to know how to set a password by leveraging the `.Description()` method and how to set the data into the SAM using the `.SetInfo()` method. You then learned how to search a system using the ADSI adapter, by leveraging the `::Exists` argument, to search for users and groups on a system.

The next section of the chapter explored windows services and processes. You started by learning about Windows Services and how to use the `get-service`, `set-service`, `stop-service`, and `start-service` cmdlets. You learnt how to change the description and the start-up parameters for Windows services. You then dived into Windows processes and got familiar with how to use the `get-process`, `start-process`, and `stop-process` cmdlets. You also understood about the different process properties of Handles, Non-paged Memory, Pageable Memory, Working Set, Virtual memory, CPUs, Processor ID, and Process name. Additionally, you learned how to search for processes on a system by leveraging wildcards and calling the `-id` parameter.

You finished this chapter learning about Windows features, and PowerShell's interaction with them. You found out about the `get-windowsfeature`, `install-windowsfeature`, and `uninstall-windowsfeature` cmdlets. You also understood how to query, install, and uninstall the Telnet-Client and Telnet-Server features. At the end of this chapter, you should have a basic understanding of managing Microsoft systems with PowerShell. In the next chapter, you will learn about automation leveraging PowerShell. This includes internally invoking scripts as well as desired state configuration.

Chapter 13. Automation of the Environment

One of the fastest growing uses for PowerShell is the automation of network environments. Whether it is automating mundane tasks or dynamically provisioning entire systems, PowerShell provides limitless options for developers to put their creative touch on system automation.

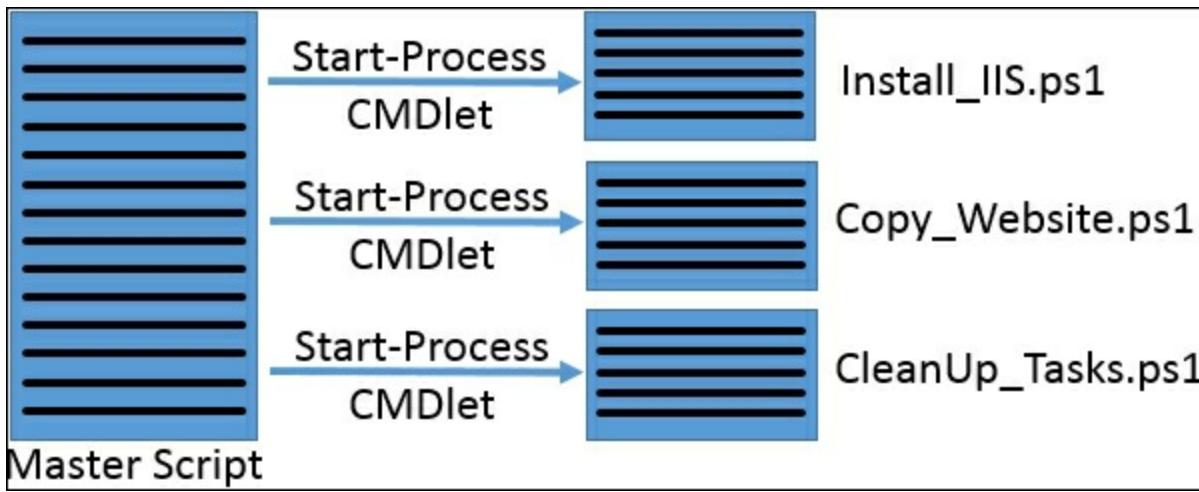
In this chapter, you will learn the following:

- Invoking programs for automation
- Using desired state configuration
- Detecting and restoring drifting configurations

Invoking programs for automation

When you want to automate the provisioning of systems, Microsoft provides many tools that enable you to execute items in a sequence. With **Microsoft Deployment Toolkit (MDT)**, **Deployment Workbench**, **System Center Configuration Manager**, **Desired Configuration Management**, and **System Center Orchestrator**, you have the ability to stage different tasks in sequential order. This allows administrators to pre-stage prerequisites on a system, before installing additional software. While these tools are extremely effective, there are instances where you may not have access to, or licensing for, the use of these products. This section explores alternative options for dynamically provisioning systems, and how to sequence a series of scripts.

The following graphic represents how you can have a parent child relationship between scripts:



The first and most important step in architecting an automation solution is to create a master script, to invoke the subsequent steps in the build process. While you could create all the steps for the automation into a giant script, it can be extremely cumbersome to troubleshoot. It is recommended that you break apart all the individual components you are automating into individual scripts, and invoke these scripts from a master file. This allows you to only troubleshoot or update the individual components, and makes management of the automation solution much easier.

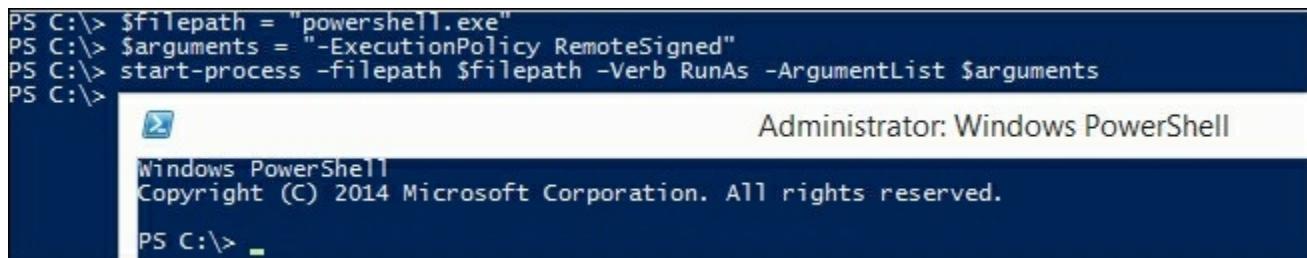
The three most popular cmdlets that are used for invoking automation through PowerShell are the `start-process`, `invoke-item`, and `invoke-expression` cmdlets. The `start-process` cmdlet is used to invoke one or multiple processes on a system. This is one of the most flexible cmdlets, which allows the most number of arguments, triggers, and verbs to execute the task at hand. The proper syntax for using this cmdlet is calling the `start-process` cmdlet, providing the `-filepath` trigger pointing to the location of the item you want to execute. You can then call the optional `-argumentlist` trigger, referencing the triggers that are needed to execute the item, the optional `-verb` trigger to invoke any verbs associated with the file type (such as `Edit`, `Open`, `Play`, `Print`, and `RunAs`), the optional `-NoNewWindow` trigger to not spawn the command in a new PowerShell window, and the optional `-wait` trigger to wait for the process to complete before continuing with the script. If you do not

execute the `-wait` trigger, the script will continue to the next step without waiting for the current step to be successful.

To launch an administrator PowerShell window, do the following:

```
$filepath = "powershell.exe"
$arguments = "-ExecutionPolicy RemoteSigned"
start-process -filepath $filepath -Verb RunAs -ArgumentList
$arguments
```

The output of this is shown in the following screenshot:



PS C:\> \$filepath = "powershell.exe"
PS C:\> \$arguments = "-ExecutionPolicy RemoteSigned"
PS C:\> start-process -filepath \$filepath -Verb RunAs -ArgumentList \$arguments
PS C:\>

Administrator: Windows PowerShell

Windows PowerShell
Copyright (C) 2014 Microsoft Corporation. All rights reserved.

PS C:\> .

The preceding example displays how to properly launch an administrator PowerShell window, with the execution policy of `RemoteSigned`, using the `start-process` cmdlet. You start by declaring the `$filepath` variable and setting it equal to `powershell.exe`. You then declare the `$arguments` variable and set it to `-ExecutionPolicy RemoteSigned`. Finally, you call the `start-process` cmdlet with the `-filepath` trigger referencing `$filepath`, the `-Verb` trigger set to `RunAs`, and `-argument` set to `$arguments`. After running the command, a new administrator PowerShell window will launch on your system. This is useful in instances where you need a PowerShell window to invoke with a specific execution policy level and under the administrator account.

The `invoke-item` cmdlet differs from the `start-process` cmdlet in the way that it is solely designed to natively open files using the default open action. The proper syntax for this cmdlet is to call the `invoke-item` cmdlet and reference a file path to an executable. If the executable that you are invoking is in the `System32` directory of the system you are executing it on, you can just reference the executable name, and it will launch accordingly.

Tip

The `invoke-item` cmdlet is deprecated and may be removed in subsequent releases of PowerShell. PowerShell natively invokes executables by calling the executable path as a line in a script, which makes the `invoke-item` cmdlet likely to go away.

To leverage `invoke-item` to launch calculator, do the following:

```
invoke-item "c:\windows\system32\calc.exe"
```

The output of this is shown in the following screenshot:

```
PS C:\> invoke-item "c:\windows\system32\calc.exe"
PS C:\>
```

The above example displays how to properly use the `invoke-item` cmdlet to launch the Windows calculator. You start by typing `invoke-item`, followed by the location of the Windows calculator of `c:\windows\system32\calc.exe`. After running this script, the Windows calculator will launch.

The `invoke-expression` cmdlet is primarily used to execute lines of code from a string. The proper syntax of this cmdlet is calling the `invoke-expression` cmdlet, and referencing a string `$variable` that contains code. This string can contain a PowerShell command or even a command-line expression. This cmdlet differs from `invoke-item` and `start-process` in the way that it natively executes command-line code, without having to call the `CMD.exe` or `PowerShell.exe` executable with the appropriate triggers:

```
$string = "ping 127.0.0.1"
invoke-expression $string
```

The output of this is shown in the following screenshot:

```
PS C:\> $string = "ping 127.0.0.1"
PS C:\> invoke-expression $string

Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

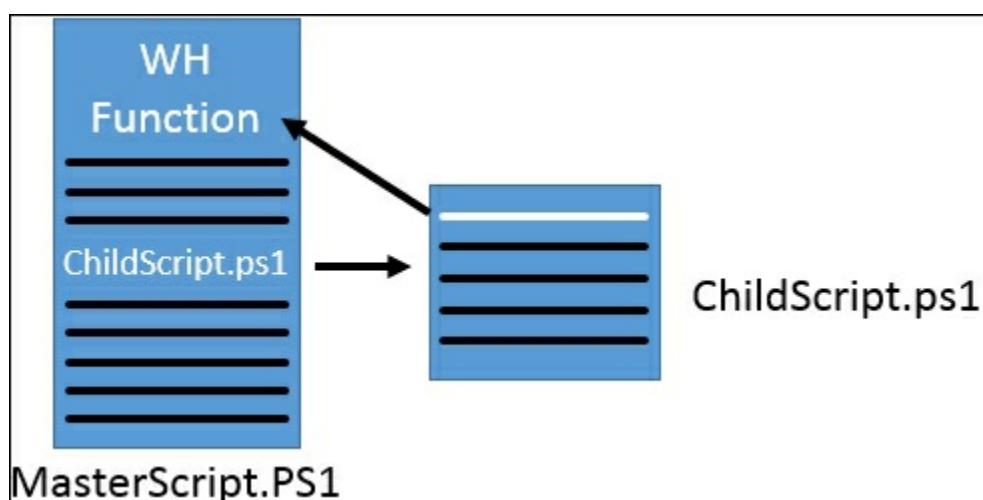
Ping statistics for 127.0.0.1:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
  Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
PS C:\>
```

This example displays how to use the `invoke-expression` cmdlet to run a command from a string. You start by declaring a variable of `$string` and setting it equal to `ping 127.0.0.1`. You then call the `invoke-expression` cmdlet with the `$string` variable. After executing this script on your system, you will see a loopback ping response in your PowerShell console.

Note

It is important to note that the `invoke-expression` cmdlet presents a security vulnerability when used in scripts. If the expression is dynamically generated, or generated as a result of user input, one could theoretically invoke a variety of actions on a server, simply by inserting code into the user-defined fields.

The following graphic represents how the variables and functions defined in a parent can be shared with the child script:



When you are chaining multiple scripts together, you have the ability to share all the variables and functions that you define in the parent script with the child script. This is a result of PowerShell sharing the same memory session between all invoked PowerShell windows from a parent window. While this makes all the items available to the child script, if you define the same named variables and functions in the child script, then the child script's version of those items will prevail. However, when the child script closes, PowerShell will dispose of the child script's version and the parent's version of the variable, method, and functions will be made available for further use.

To show how scripts can share functions, you will need to stage your computer with two files. To do this, you will need to follow these steps:

1. Create the directory of `c:\temp\scripts\` (if it doesn't already exist).
2. Create a PowerShell file in `c:\temp\scripts\` named `masterscript.ps1`. Place the following code in `masterscript.ps1`:

```
function wh { param([string]$message)
    write-host "Wh Function Output is: $message"
}
write-host "MasterScript.ps1: Launching Child Script..."
invoke-expression -command c:\temp\scripts\childscript.ps1
```

3. Create a PowerShell file in `c:\temp\scripts\` named `childscript.ps1`. Place the following code in `childscript.ps1`:

```
wh "From ChildScript.ps1: The wh Function resides in
MasterScript.ps1 file."
wh "From ChildScript.ps1: Childscript.ps1 is accessing the
wh Function successfully from memory."
pause
```

After both the files have been created, you leverage the `Invoke-Expression` cmdlet with the `-Command` trigger set to `c:\temp\scripts\masterscript.ps1` to launch the `masterscript.ps1` file.

To run the `masterscript.ps1` file, which will execute the `childscript.ps1` file, do the following:

```
invoke-expression -command c:\temp\scripts\masterscript.ps1
```

The output of this is shown in the following screenshot:

```
PS C:\> invoke-expression -command c:\temp\scripts\masterscript.ps1
MasterScript.ps1: Launching Child Script...
Wh Function Output is: From ChildScript.ps1: The wh Function resides in MasterScript.ps1 file.
Wh Function Output is: From ChildScript.ps1: Childscript.ps1 is accessing the wh Function successfully from memory.
Press Enter to continue...:
```

This script displays how to share a function between a parent and child script. You start by manually invoking the script by using the `invoke-expression` cmdlet, using the `-command` parameter with the argument of `c:\temp\scripts\masterscript.ps1`. After executing this script, the `masterscript.ps1` file will print to the screen `MasterScript.ps1: Launching Child Script...` and launch the `childscript.ps1` file. The `childscript.ps1` file will then invoke the `wh` function with text overloads in the `masterscript.ps1` file.

The console will then print to the screen `Wh Function Output is: From ChildScript.ps1: The wh Function resides in MasterScript.ps1 file.` and `Wh Function Output is: From ChildScript.ps1: Childscript.ps1 is accessing the wh Function successfully from memory.` This properly displays the `childscript.ps1` file, successfully utilizing the `wh` function defined in `masterscript.ps1` file.

As you are building a series of scripts to automate your environment, you can share the large repeating functions across your scripts to reduce their overall size. This not only reduces the complexity of the individual scripts, but requires you to only update the large repeating functions in one location as you improve them. This creates greater efficiencies with scripting and will produce more reliable scripts.

Using desired state configuration

Desired state configuration (DSC) is a management layer which enables you to dynamically build computer environments using a common language. This management layer was designed to create *configuration baselines*, which enforce a set of configuration standards for a system. Simply put, it makes sure that the files, folders, services, registry, and applications that you expect to be installed or removed from a system, are in their expected state. If DSC determines that these items are not in their *desired state*, it will automatically correct the system to ensure it follows the *desired configuration*.

There are three core phases for a proper implementation of DSC. These phases include:

- **Authoring phase:** The authoring phase is the process where you create the configuration. The configuration will have a subset of configuration items such as files, folders, services, registry, and applications. This is where you create *what is desired* to be configured for a particular configuration.
- **Staging phase:** The staging phase is the process by which the system compares the desired state against the current running configuration. DSC then determines what items in the configuration need to be enforced. The files are then staged to be pushed to a server or pulled from a server.
- **Remediation phase:** The remediation phase is the enforcement state. It updates the configuration of a system, so that it matches the desired configuration.

The authoring phase enables you to create a configuration, where you can specify any number of resources for configuration of a system. PowerShell's 4.0 implementation of DSC has 12 resources that you can configure, to automate and enforce the installation of items on a system.

These resources available to DSC include:

- **Archive resource:** This resource is responsible for unzipping files in

a specified path during the configuration. The mandatory properties for this resource are `Ensure`, `Destination`, and `Path`. The optional properties are `Checksum`, `DependsOn`, `Validate`, and `Force`.

- **Environment resource:** This resource is responsible for modifying the environment variables during the configuration. The mandatory properties for this resource are `Ensure`, `Path`, and `Value`. The optional property is `DependsOn`.
- **File resource:** This resource is responsible for managing files and folders during the configuration. The mandatory properties for this resource are `Type`, `Ensure`, `SourcePath`, and `DestinationPath`. The optional properties are `Recurse`, `Attributes`, `Checksum`, and `DependsOn`.
- **Group resource:** This resource is responsible for configuration of local groups during the configuration. The mandatory properties for this resource are `Ensure` and `GroupName`. The optional properties are `Credential`, `Description`, `Members`, `MembersToExclude`, `MembersToInclude`, and `DependsOn`.
- **Log resource:** This resource is responsible for logging the information pertaining to the configuration, on the system during the configuration. The mandatory property for this resource is `Message`. The optional property, almost always used with the log resource, is `DependsOn`.
- **Package resource:** This resource is responsible for working with software installation packages such as MSI's and `setup.exe` during the configuration. The mandatory properties for this resource are `Ensure`, `Path`, `Name`, and `ProductID` (for MSI's). The optional properties are `Arguments`, `Credential`, `LogPath`, `DependsOn`, and `ReturnCode`.
- **WindowsProcess resource:** This resource is responsible for working with Windows processes during the configuration. The mandatory properties for this resource are `Ensure`, `Arguments`, and `Path`. The optional properties are `Credential`, `StandardErrorPath`, `StandardInputPath`, `StandardOutputPath`, `WorkingDirectory`, and `DependsOn`.
- **Registry resource:** This resource is responsible for working with the registry during the configuration. The mandatory properties for

this resource are `Ensure`, `Key`, and `ValueName`. The optional properties are `Force`, `Hex`, `ValueData`, `ValueType`, and `DependsOn`.

- **WindowsFeature resource:** This resource is responsible for working with the Windows features during the configuration. The mandatory properties for this resource are `Name` and `Ensure`. The optional properties are `Credential`, `IncludeAllSubFeature`, `LogPath`, `Source`, and `DependsOn`.
- **Script resource:** This resource is responsible for executing PowerShell script blocks during the configuration. Depending on usage, the mandatory properties of this resource may be `SetScript`, `GetScript`, or `TestScript`. The optional properties are `Credential` and `DependsOn`.
- **Service resource:** This resource is responsible for managing the Windows Services during the configuration. Depending on usage, the mandatory properties for this resource are `Name` or `State`. The optional properties are `BuiltInAccount`, `Credential`, `StartupType`, and `DependsOn`.
- **User resource:** This resource is responsible for configuration of local users during the configuration. The mandatory properties for this resource are `UserName`, `Ensure`, and `Password`. The optional properties are `Description`, `Disabled`, `Fullname`, `PasswordChangeNotAllowed`, `PasswordChangeRequired`, `PasswordneverExpires`, and `DependsOn`.

Each of these resources has optional and mandatory properties. The mandatory properties must be set in your configuration, or the resource will not properly configure. Properties define the action that needs to be performed for that resource.

Tip

For more information on the properties for each of these resources, you can view the Microsoft TechNet article at
<http://technet.microsoft.com/en-us/library/dn249921.aspx>.

Authoring phase

The four required components to author a new DSC configuration item are declaring a name for the configuration, defining the nodes, referencing configuration resources, and setting the properties for those resources.

The following code represents the proper syntax of a DSC configuration item:

```
configuration InstallTelnet {
    param($computers)
    Node $computers {
        WindowsFeature Telnet-Client {
            Name = "TelnetClient"
            Ensure = "Present"
            IncludeAllSubFeature = "True"
        }
    }
}
InstallTelnet -computers MyComputer
```

The name for the configuration is called much like a function in PowerShell. You start by declaring `configuration` `NameOfConfiguration {`. The name of configuration should represent the overall configuration you are trying to achieve. In the preceding example, you are going to install telnet, so you name the configuration `InstallTelnet`.

You may also wish to define the optional `param()` code block to define parameters for the configuration item. In the previous example, you created a `param()` code block with the `$computers` parameter to specify the computers for the node component.

The node component specifies which systems the configuration will generate the MOF files for. To specify the nodes, you use the syntax of `node Value {`. The `Value` parameter, specified in the syntax for the `node` component, can contain a string, a variable, or an array. In the preceding example, you leverage an array named `$computers`, for which individual MOF files will be created.

The next step is to call the individual configuration resources. This is

done by calling configuration resource such as `WindowsFeature Telnet-Client {`. The `WindowsFeature` portion tells PowerShell you are leveraging the `WindowsFeature` resource. When you specify `Telnet-Client`, it is the reference name of that particular configuration resource. The name of the configuration resource can be referenced by other parts of your script for properties like `DependsOn`. This means that you can setup dependencies on other configuration resources, so that items install in order.

After you define the configuration resource, the last component requires you to define its mandatory properties. The proper syntax for adding properties is `propertyname = Value`. The property name must reflect an actual mandatory or optional property for that resource. Subsequent properties for the resource can be declared on an additional line below the other properties. In the preceding example, there are three mandatory properties defined. The `Name` property is set to `TelnetClient`, the `Ensure` property is set to `Present`, and the `IncludeAllSubFeature` is set to `True`. You then close the bracket for configuration resource, node, and the configuration item itself. At this point, you have successfully created a configuration item.

After creating the configuration items, proceed to execute the configuration to create MOF files for the individual nodes. During this phase, the configuration is validated against all nodes that will receive the packages. To start the validation, you call the name of configuration item of `NameOfConfiguration`, and then any of the variables you want to include in the `param()` block for the configuration item. In the previous example, you call the configuration item of `InstallTelnet` and specify the `-computers` parameter with the `MyComputer` argument.

After calling the configuration item, the system will automatically generate a folder with the configuration item name as the folder name. Inside that folder, you will find a **MOF** file named after every node. This file is used during the staging and remediation phase to change the systems.

If you want to create a configuration item for `Telnet-Client` and

Telnet-Server, do the following:

```
configuration InstallTelnet {
    param($computers)
    Node $computers {
        WindowsFeature Telnet-Client
        {
            Name = "Telnet-Client"
            Ensure = "Present"
            IncludeAllSubFeature = "True"
        }
        WindowsFeature Telnet-Server
        {
            Name = "Telnet-Server"
            Ensure = "Present"
            IncludeAllSubFeature = "True"
            DependsOn = "[WindowsFeature]Telnet-Client"
        }
    }
}
$computer = $env:computername
InstallTelnet -computers $computer
```

The output of this is shown in the following screenshot:

PS C:\scripts> configuration InstallTelnet {
>> param(\$computers)
>> Node \$computers {
>> WindowsFeature Telnet-Client
>> {
>> Name = "Telnet-Client"
>> Ensure = "Present"
>> IncludeAllSubFeature = "True"
>> }
>> WindowsFeature Telnet-Server
>> {
>> Name = "Telnet-Server"
>> Ensure = "Present"
>> IncludeAllSubFeature = "True"
>> DependsOn = "[WindowsFeature]Telnet-Client"
>> }
>> }
>> }
>>
PS C:\scripts> \$computer = \$env:computername
PS C:\scripts> InstallTelnet -computers \$computer

Directory: C:\scripts\InstallTelnet

Mode	LastWriteTime	Length	Name
-a--	12/31/2014 2:43 PM	1910	DEMODC1.mof

This example displays how to properly leverage **Desired State Configuration** and the `WindowsFeature` resource, to create a MOF file for installing the Telnet client and server on a system. You start by declaring the configuration item of configuration `InstallTelnet {`. You then accept in a parameter of `$computers` leveraging the `param($computers)` line of code. The next step is declaring the `node`, which will be the content of the parameter, which would be `Node $computers {`. You then declare the `WindowsFeature` resource and `reference Telnet-Client WindowsFeature`. To complete the configuration of `WindowsFeature`, you set the `name` property to `Telnet-Client`, the `Ensure` property to `Present`, and the `IncludeAllSubFeature` property to `True`.

You continue by declaring another `WindowsFeature` resource and `reference Telnet-Server WindowsFeature`. To complete the configuration of `WindowsFeature`, you set the `name` property to `Telnet-Server`, the `Ensure` property to `Present`, the `IncludeAllSubFeature` property to `True`, and the `DependsOn` property to `[WindowsFeature]Telnet-Client.`

You then leverage the `$env:computername` function and set it to the `$computer` variable. You complete this script by calling the configuration item of `InstallTelnet`, the `-computers` trigger, followed by the `$computer` variable. After execution, you will find that there will be a folder named `InstallTelnet`, with a MOF file named as `computername.MOF`.

Staging and remediation phase

The two different types of deployment mechanisms that DSC supports for configuring systems are pull and push types. The main difference between pull and push methodology is where the command is being processed. In the pull methodology, you set up a pull server like **System Center Configuration Manager (SCCM)** or a PowerShell DSC Pull Server, and it will synchronize with the appropriate clients for the

configuration. A pull server often offers built-in reporting of compliance, and can schedule a time where the configuration item can be pulled from the server.

With the push method, you are leveraging the `start-dscconfiguration` cmdlet to push the configuration, in real-time, to the specified nodes. The proper syntax for the cmdlet is referencing the `start-dscconfiguration` cmdlet, specifying the `-path` trigger, referencing the location where the MOF files were created, and referencing the configuration item name. You may also specify the `-wait` trigger to wait for that configuration to complete before proceeding to the next line of code, the `-force` trigger to override any confirmation of changes, and the `-verbose` trigger to print to the screen the verbose output from PowerShell, configuring the individual systems.

The remediation phase is where the configuration item hits the local system, is evaluated, and is applied to a system. The remediation phase is complete after the configuration item successfully configures a system. This is the phase when you detect if a configuration has drifted from the desired state. It will reapply the configuration to ensure compliance to that desired state.

To push a configuration item to a local system, do the following:

```
start-DscConfiguration -path .\InstallTelnet -wait -force
```

The output of this is shown in the following screenshot:

```
PS C:\scripts> start-DscConfiguration -path .\InstallTelnet -wait -force
```

This example displays how to properly leverage the push methodology to configure systems in an environment. After generating the MOF file, you are ready to make the configuration item changes. You start by calling the `start-dscConfiguration` cmdlet, with the `-path` trigger pointing to the root location of where the MOF files were generated. In this case, it is `.\InstallTelnet`. You then declare the `-wait` trigger, so

the script doesn't continue to the subsequent lines in the script until Telnet is fully installed. Last, you issue the `-force` trigger to suppress any confirmation messages. After execution, the server will have both the Telnet-Server and Telnet-Client Windows features installed on it.

Tip

It is important to note that after you run the `start-dscConfiguration` cmdlet, the MOF files are still present on a system after execution. You may want to consider deleting these files after execution. The MOF files are created in clear text and may contain sensitive information about the system you are configuring.

Detecting and restoring drifting configurations

PowerShell's integration with DSC provides the ability to evaluate the current state of the desired configuration on a system. After the configuration has been set on the system, you have the ability to check the current configuration using the `get-dscconfiguration` cmdlet. By simply calling `get-dscconfiguration`, the PowerShell cmdlet will evaluate what DSC items have been designated for the system, and how they were configured.

You may also want to determine if the system has drifted from the existing configuration set. Drifting happens when engineers modify a system, or additional software changes the configuration of that server. To test the existing configuration set, you can leverage the `test-dscconfiguration` cmdlet. The `test-dscconfiguration` cmdlet will either return `True`, which means that the system is configured as desired, or `False`, which means the system no longer adheres to the desired configuration for the system.

When you determine that a configuration has drifted from the desired state of configuration, you have the ability to restore the original configuration itself. This is done through leveraging the `restore-dscconfiguration` cmdlet. Often, `test-dscconfiguration` is used in conjunction with the `restore-dscconfiguration` cmdlets, as you can immediately remediate issues if the test returns `False`.

The `get-dscconfiguration`, `test-dscconfiguration`, and `restore-dscconfiguration` cmdlets support `cimsessions`. You can declare a `cimsession` using the `new-cimsession` cmdlet to query multiple systems configuration. By using the optional `-cimsession` trigger with the `get-dscconfiguration`, `test-dscconfiguration`, and `restore-dscconfiguration` cmdlets, you can retrieve information and reconfigure items over a session for remote systems.

get-dscconfiguration

```
PS C:\scripts> get-dscconfiguration

Credential          : Telnet Client
DisplayName        : Telnet Client
Ensure             : Present
IncludeAllSubFeature : False
LogPath            :
Name               : Telnet-Client
Source             :
PSComputerName    :

Credential          : Telnet Server
DisplayName        : Telnet Server
Ensure             : Present
IncludeAllSubFeature : False
LogPath            :
Name               : Telnet-Server
Source             :
PSComputerName    :
```

In this example, you are querying a system that you set a desired state configuration item on, using the `get-dscconfiguration` cmdlet. After executing the command, you will see the expected desired state configuration printed to screen. You will also see the expected configuration of those individual resources on the system. In this case, you see that the DSC resources of `Telnet-Client` and `Telnet-Server` are expected to be `Present` in the system:

test-dscconfiguration

```
PS C:\scripts> test-dscconfiguration
True
PS C:\scripts>
```

In this example, you are querying a system that you set a desired state configuration item on, using the `test-dscconfiguration` cmdlet. After executing the command, you will see that the existing configuration matches the expected configuration with the cmdlet returning `True`:

```
Remove-WindowsFeature -name Telnet-Server
write-host "Telnet-Server Feature Has Been Manually Removed
From The System"
$testresult = test-dscconfiguration
```

```

if ($testresult -like "False") {
    Restore-dscconfiguration
}
$testresult = test-dscconfiguration
if ($testresult -like "True") {
    write-host "Telnet-Server Successfully Restored on The
System"
}

PS C:\> Remove-WindowsFeature -name Telnet-Server
Success Restart Needed Exit Code      Feature Result
----- ----- ----- -----          -----
True   No           Success          {Telnet Server}

PS C:\> write-host "Telnet-Server Feature Has Been Manually Removed From The System"
Telnet-Server Feature Has Been Manually Removed From The System
PS C:\> $testresult = test-dscconfiguration
PS C:\> if ($testresult -like "False") {
>>     restore-dscconfiguration
>> }
>> $testresult = test-dscconfiguration
>> if ($testresult -like "True") {
>>     write-host "Telnet-Server Successfully Restored on System"
>> }
>>
WARNING: [DEMODC1]: [[WindowsFeature]Telnet-Server] Windows automatic updating is not
enabled. To ensure that your newly-installed role or feature is automatically updated, turn on Windows Update.
Telnet-Server Successfully Restored on System

```

This example displays how to restore a configuration when a system drifts from the desired state. You start by executing a manual command to remove Telnet-Server from the system. This is done by using the Remove-WindowsFeature cmdlet with the –name trigger set to Telnet-Server. You then print to the screen Telnet-Server Feature Has been Manually Removed From The System.

Thereafter, you would query the existing configuration to determine if it has drifted from the initial configuration, using the test-dscconfiguration cmdlet and storing the result in the \$testresult variable. Next, you create an IF statement to determine if the \$testresult variable is set to False. If it is set to False, it means the configuration has drifted, and you execute the restore-dscconfiguration cmdlet to restore it to the desired state. After the restore-dscconfiguration cmdlet is done executing, you re-execute test-dscconfiguration and store it in \$testresult to determine if the configuration has been restored. If the \$testresult variable is set to True, you then print to the screen Telnet-Server Successfully

Restored on System. After executing the script, you manually remove Telnet-Server, validate the drift in configuration, and restore Telnet-Server back on the server using Desired State Configuration.

Summary

This chapter explained many different facets of automation in environments. You started by learning how to invoke programs for automation. You learned about the cmdlets `start-process`, `invoke-item`, and `invoke-expression`, and how they can be used to initiate actions on a system. Next, you understood how to chain multiple PowerShell scripts together in a child and parent relationship. You also learnt how to share functions, methods, and variables between multiple scripts, and how to invoke them between each other.

The chapter proceeded to dive into DSC and explained about its three phases, and the variety of resources you can apply to your configurations. You also learned how to detect a drifting configuration and how to restore it if it drifts from the desired state. In the next chapter, you will learn some of the recommended best practices for PowerShell. These best practices will help create efficiencies in your scripts and make your scripts portable between engineers.

Chapter 14. Script Creation Best Practices and Conclusion

This chapter explores the best practices for script creation. It begins by discussing the ways to create comment headers in your scripts. It then dives into how to comment on code in your scripts and different situations where you may need detailed comments. You will then explore the best practices for script creation and provide other considerations that developers need to have in order to develop their scripts. This chapter then explains source control and maintaining revisions, and concludes with the best practices for automation with PowerShell.

Best practices for script management

When you are creating and maintaining scripts, you may create multiple iterations of your scripts. Whether it is bug fixes or adding new functionality, it's sometimes difficult to keep track of all of the different versions of your code. This section explores tips for better managing your scripts and provides insight into tricks that you can do to make your iterations more reliable.

commenting headers

The first recommendation is to create headers for the detailed tracking information about the PowerShell script itself. Headers can track information about the script's creation, authors, changes, and other useful information that will enable you to quickly determine what the script is doing. PowerShell has built-in block comment support, which integrates with the `get-help` cmdlet.

The required components for this include:

- **Comment block location:** The comment block must be the first item defined at the top of your script. If you use parameter blocks, you will need to specify the parameter blocks after the comment block.
- **Start comment block:** In order to integrate with the help system, you need to specify the starting of the comment block. To start a comment block, you type `<#`.
- **.SYNOPSIS:** To create a synopsis for the script, type `.SYNOPSIS` on a line and then on a subsequent line, type a one line description of what the script is for.
- **.DESCRIPTION:** To create a full description for the script, type `.DESCRIPTION` on a line and then on a subsequent line, type a description of the script's functions so that any editor who looks at the script will know the script's basic functions. If it is a complicated script or a script that invokes other scripts, describe the overall process for the script. You may also want to include author information such as the author's name, author's position, author's company and contact information, initial release number, and date of the initial release.
- **.PARAMETER:** To describe a parameter for the script, you can type `.PARAMETER`, the parameter name on a line and then on a subsequent line, type a description of that parameter. If you have multiple parameters, you can define multiple `.PARAMETER` statements referencing parameter names.
- **.EXAMPLE:** To provide an example of usage for your script, you

can type .EXAMPLE and on a subsequent line, type a usage example of the script. If you have multiple examples, you can define multiple .EXAMPLE statements.

- **.NOTES:** To provide notes for execution caveats, you can type .NOTES and on a subsequent line, type a usage note to execute the script.
- **.LINK:** If you have other help topics you want to link to, you can type .LINK and on a subsequent line, provide a URL to another help topic.
- **Ending comment block:** In order to integrate with the help system, you need to specify the ending of the comment block. To end a comment block, you need to type #>.

The following graphic displays a properly commented header that will integrate with the help system:

```
<#
.SYNOPSIS
This script scans the network file shares for clear text files, and determined if there
are clear text username and passwords in the files.

.DESCRIPTION
This script scans the network file shares for clear text files, and determined if there
are clear text username and passwords in the files.

Author: Brenton J.W. Blawat / Packt Publishing / Author / email@email.com
Revision: 1.3a - Initial Release of Script / 5-27-2016
Revision: 1.4 - Paul Brandes / Company XYZ / Consultant / email@company.com / 10-27-2016
R1.4 Details: Corrected the scanning function to accommodate large UNC paths and process
paths over 248 character

.PARAMETER PATH
The optional path parameter enables you to specify a path to the file structure to scan.

.EXAMPLE
powershellscript.ps1 /path \\uncpath\folder\

.NOTES
You must have administrative rights to the paths you are scanning. You must run the script
in an Administrator powershell window.
#>
```

This properly shows the syntax of a comment block that is usable via the PowerShell help system. You start by defining the comment block of <#. You then create a .SYNOPSIS section with a synopsis of the script. You then create a .DESCRIPTION section and provide a detailed description of the script. You also provide the author, revision, and editing revisions to

the script. Following that section, you define a `.PARAMETER PATH` section, which provides detailed information about the `PATH` parameter in the script. You create an `.EXAMPLE` section to provide an example for usage, and you create a `.NOTES` section to provide information about the required execution environment. You then close the comment block by issuing `#>`.

Commenting code

The second recommendation is to keep track of changes in the PowerShell scripts in-line with the code. While most developers are great at providing comments pertaining to the overall functions, there are some guidelines that should be followed for commenting code in-line with the code, which are given here:

- **Comment the usage of all functions:** As you are developing functions to use within your scripts, it is recommended that you comment on how to use these items. This should include all mandatory parameters and all optional parameters being fed into the function. You should also specify the input and the output of the function.
- **Comment bug fixes:** When you are creating code to fix known bugs with the script or bugs within the system you are trying to configure, it is helpful to document the bug properly. Create a comment that references a bug number, TechNet article, or URL to provide insight into why you are performing code that might seem out of place in the script.
- **Comment backwards compatibility:** It is very common for environments to have a mixed variety of PowerShell versions in the environment. When you are creating code to provide backwards compatibility, it is recommended that you fully comment on what you're engineering for previous versions of PowerShell. This will provide insight into why you are not using a more efficient way of coding the script through new cmdlets.
- **Comment complex math calculations or formulas:** There may be instances where you need to leverage complex equations for data within PowerShell. It is recommended that you fully comment on

the equations so that others can understand how you are deriving the data.

- **Comment the third-party modules:** When you are importing third-party modules for use in your PowerShell scripts, it is recommended that you provide the URL of the location where you downloaded the modules and other details about the module. This can include how you are using it, why you are using it, and where licensing information is for the third party module.
- **Comment .NET references:** In instances where you need to call .NET reflection assemblies to perform code, it is recommended that you comment on the individual lines of the code. Since most PowerShell users don't have a traditional .NET development background, it's important to comment on the individual lines of PowerShell code using .NET objects and what they are used for.
- **Comment in third person present tense:** When you are creating comments for your scripts, always try to create all comments in third person and present tense. This will keep the code more factual to *what is* being implemented instead of *what was* implemented. It also provides the developers the opportunity to identify themselves inside the comments, which is easier when non-identifying *I or me* words are not used. Only in cases of backwards compatibility, it is acceptable to use the past tense for historical information.

Best practices for script creation

As you are developing your scripts, there are standardized best practices that should be followed during the coding stage. These best practices provide robust and structured scripts, which will have predictable results. This section explores some of the recommendations for efficient PowerShell coding and provides guidance on how to avoid programming headaches.

Script structure

When you are creating scripts, you should adhere to a strict script structure. The structure of the script dictates the order of execution and how things are processed. It is recommended that you structure your scripts as shown here:

- 1. Declare the script header:** To start the scripting process, declare a header and include everything that is pertinent to the script. This may include a description, revision information, author, editor, and additional notes.
- 2. Declare the input parameters:** After the header, declare your input parameters if required by the script. Not only is this necessary, but it helps developers identify what is being input into the script and what fields are required for proper exaction of the script.
- 3. Declare the global variables:** You should declare your global variables after the input parameters. As the variables are declared in a function, only stay within the boundary of function; the variables you need to use globally should be defined here.
- 4. Declare the functions in order:** The next portion of the structure is declaring your functions. If you have functions that call each other, you will want to declare the functions that are called first in the script. This ensures that your scripts will not error out due to the function not being declared prior to execution.
- 5. Start the execution of script:** After declaring the functions, you can start the execution of the script. This section will call the other functions that use the global variables and parameters to complete

the tasks for the script.

6. **Declare the end of the script:** After execution, it is recommended that you create an indicator that specifies that the script runs successfully. Whether it is logging an exit code to a file or pausing the script at the end, it is important for you to create logic declaring proper or improper execution of your script.

Other important best practices for script creation

In addition to creating a proper script structure, there are a few other recommendations that should be considered when you are developing your scripts. These items include:

- **Limiting the use of cmdlet aliases:** While aliases provide the ability to shorten the overall size of a script, they also add complexity for those who are not familiar with a particular alias. This can also cause problems while editing scripts you haven't visited in a while if you have forgotten what the alias relates to. Another reason why you should avoid aliases is because they are not always portable from one machine to the next. This can be due to conflicting PowerShell versions or because the aliases are not defined on other machines. It's recommended to use aliases only during testing and manual PowerShell interactions.
- **Know when and when not to use a script:** While PowerShell is an excellent tool for automation and management, there are times where it may not be in your best interest to create a script. In the instance of software deployment tools, instead of executing a PowerShell script that invokes a MSIEXEC installation, it's sometimes easier to just place the MSIEXEC installation command in the deployment tool itself. This reduces the number of steps that are required to determine what task is being completed. Always take time to determine if a PowerShell script is required for the task at hand.
- **Never assume that a path is available:** When you are working with PowerShell, one of the most useful tools is the `test-path` cmdlet.

This cmdlet allows you to validate files, folders, and registry items on systems before using them. Always plan for the worst case scenario and build the logic into your script to `test-path` before using files, folders, and registry items.

- **Never statically set up information in a script:** As you are developing your scripts, it is easy to statically set up information in your scripts. This information may include locations to a file, folder, registry, or usernames and passwords. Unless it's absolutely required for script execution, it is recommended that you leverage relative paths, systems variables, answer files, and prompts for usernames and password.
- **Use answer files for script static information:** When you are scripting, it's important to keep the PowerShell scripts as generic as possible. For reusability. When you need to feed information into a script, it is recommended that you use answer files that contain all of the information needed to process the script. These answer files may contain items such as path locations, domain credentials, a list of computers, and other items that would otherwise be static in a script.
- **Encrypting usernames and passwords:** In cases where you need to provide credentials to resources in your scripts, it is recommended that you always encrypt the strings when they are declared in the scripts or answer files. When you need to use them, you can decrypt these strings on the fly and store the usernames and passwords in a variable (memory). When the PowerShell script closes after execution, the PowerShell garbage collector will clear the variables from memory, and the usernames and passwords will no longer be decrypted on the system.

Note

For more information on encrypting and decrypting strings, you can view the following Microsoft TechNet article:

<https://gallery.technet.microsoft.com/scriptcenter/PowerShell-Script-410ef9df>

- **Segments of code being reused belong to a function:** As you are developing your code, you may find sections where you repeat multiple lines of code. As a best practice rule of thumb, if you need to repeat a task multiple times, create a new function to process the work. This not only keeps the length of your script down, but it also ensures that any code changes to that code can be found in one location instead of in multiple locations in your script.
- **Display progress indicator on the screen:** For most PowerShell operations, you should be able to calculate a progress percentage to leverage the `write-progress` cmdlet. This provides the end users with the ability to approximately gauge the progress of the script. In instances where you are running the script and it's difficult to update the progress percentage, it is recommended that you leverage the `write-debug` cmdlet to print a status to the screen. This way, you can give the script user an approximate duration of the operation that is occurring on the screen.

Note

More information on the `write-progress` cmdlet can be found at <https://technet.microsoft.com/en-us/library/hh849902.aspx>.

- **Always use exception handling:** Exception handling is one of the most important components to be learned when becoming a PowerShell expert. Typically while scripting, 40 percent of your time goes to making the script work as intended and the remaining 60 percent is catching scenarios of the script failing. Whether it is validating the data extracted in the script, the files, folders, and registry items that exist before use, or to see if your changes were successful, it is essential for you to leverage exception handling.
- **Limit the use of regular expressions:** While regular expressions are very helpful in validating data, there is a large learning curve. Limit the use of regular expressions in your scripts. When you need to use them, fully comment them so others can quickly understand their function in your scripts.
- **Use the invoke-expression cmdlet cautiously:** There are well known security implications of leveraging the `invoke-expression`

cmdlet in your scripts. If you need to leverage `Invoke-Expression`, reduce the privileges account running the block of code and ensure that all part of the expression cannot be modified by user input.

Controlling source files

It is well known in the industry that you should be using source control on all code that you are developing. In the industry, however, many developers ignore the need for source control for network, batch, VB, and PowerShell scripts, as they can be easily recreated. PowerShell is developing as a language, and the need for source control will be growing as the technology evolves. There are some general recommendations for controlling the source files while you are developing. Some of these items include:

- **Use source control software (if available):** If there are source control systems that are available in your organization, you should strongly consider using them. This allows you and multiple other individuals to check out and add to the PowerShell scripts. It also allows you to track changes between different iterations of the code. These source control tools have many built-in features that can expedite the development of large and complex scripts.
- **Use Secured cloud storage systems:** An easy method to continually have a backup copy of your files is to leverage secured cloud storage systems. Some of these cloud systems also include a roll-back feature that provides source revision backups on the fly.
- **Revision major releases:** When you complete scripts for use in the environment, you should generate a new release number for those scripts. Archive the existing release and build a new folder structure for the next iteration of the scripts. This ensures that you are able to roll back to the previous versions of the script with minimal effort.

Best practices for software automation

Automation through PowerShell can be a very powerful tool when it's done right. There are few things that you can do, however, they will reduce the complexity of large automation scripts and increase reliability of execution. This section explores two things you can do that will greatly improve your success with automation when using PowerShell.

The first consideration is breaking apart scripts that contain 2000+ lines into smaller scripts. This allows you to better troubleshoot what items are failing and also makes the scripts more reusable for different tasks in your environment. For most purposes, you can break your large scripts into multiple scripts by what you're trying to accomplish. If you are installing a large software system, you could break the script into the prerequisite software scripts and main software scripts. If you are configuring different Windows features, you could break each individual feature installation and configuration into a separate script. The segmentation of code will help your productivity and will help you quickly ease the learning curve of complex automation operations.

While breaking your scripts into smaller scripts is recommended, it is also recommended that you don't create so many scripts that the automation becomes unmanageable. If you find yourself creating 20 or more scripts for an automation task, you should try to consolidate the scripts into more manageable sections.

Note

In instances where you reuse large segments of code in all of your scripts, you may consider writing your own PowerShell module. For more information on PowerShell modules, you can go to [https://msdn.microsoft.com/en-us/library/dd878297\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/dd878297(v=vs.85).aspx).

In addition to breaking apart large scripts, the second consideration is

including four sections in your automation scripts. These sections are:

- **Prerequisite check:** This section of the script will check the system for prerequisite hardware and software items. It should include a check for items such as running or available services, installed software, or available resources such as the hard drive or memory. This ensures that the system is ready to proceed with the automation tasks.
- **Pre-installation tasks:** This will remediate anything in the prerequisite check that is not satisfactory. This could include starting and stopping services, backing up the system files or registry prior to installation, installing prerequisite software, and checking for pending restarts.
- **Installation:** This section of the script should install or configure the items that you desire on the system. This may include registry entries, executing MSI files, copying files and folders to the system, or even executing batch operations.
- **Post-installation tasks:** This section of the script will execute all cleanup tasks for the installation. This may include cleaning up after the prerequisites and the installation or starting services that were stopped in the previous steps.

While following these considerations for creation of scripts, you will have a much more consistent approach to script creation. This approach ensures that you check to make sure the script won't take down a system, you are able to update the system as intended, and you are leaving the system in a stable state after execution.

It is also important to note that some scripts may not require all of the sections. If the script you are creating doesn't require all of the sections, just leave the sections commented out or remove them entirely from the script. Use your best judgment in creating the scripts with the complexity that you need for the task at hand.

Summary

This chapter explored multiple best practices for developing scripts in PowerShell. It started by explaining how you should have commented headers in all of your scripts. It also explained that the headers should have highlighted script headers, a brief description, script author information, revision information, and usage information. It then displayed a suggested header, formatted using those best practices.

You then learned about commenting on code and different guidelines to follow for that. You proceeded to learn about the best practices for script creation. You then explored several recommendations for controlling source files. This chapter ends by explaining the best practices for automation. You learned that when you are automating large systems, you should split the scripts apart into smaller, more manageable scripts.

Mastering Windows PowerShell Scripting – conclusion

PowerShell is quickly becoming the language of choice to support Microsoft systems in organizations. With Microsoft's deep integration of PowerShell in their products, PowerShell knowledge will be a required skill set in the years to come. Module 3 has taught you the fundamentals of PowerShell scripting and is geared towards real world scenarios of how Fortune 500 companies leverage PowerShell.

In the beginning chapters, you learned the basics of PowerShell. You started by learning variables, hashes, and arrays. You also learned the data parsing, manipulation, and comparison operators. You then attained the knowledge of how to use functions, switches, and loop structures. You proceeded with working with more complex items such as regular expressions, error and exception handling, and session-based remote management.

The second half of the module was focused on managing different components of the Microsoft operating system. You started by mastering how to manage files, folders, and registry items. You then learned how to control permissions with access control lists and applying basic and advanced attributes to files and folders using PowerShell. You then dived into Windows Management Instrumentation, XML manipulation, Windows processes, Windows services, and local users and group management.

The last section of this module explained different PowerShell automation techniques and taught you how to use the desired configuration management to create and enforce configuration baselines. It concluded with providing a list of best practices and final recommendations for scripting with PowerShell.

Staying connected with the author

As the author of Module 3, Brenton Blawat always extends a helping hand to the community. You can follow Brenton on twitter @brentblawat or his blog at <http://www.bittangents.com>. The author is always open for discussions on the module and will provide feedback to readers, as time allows. To ask Module 3 related PowerShell questions, you can visit <http://www.masteringposh.com>.

If you found this course helpful, the author also encourages you to leave your feedback on the site where you purchased the course. Every comment helps spread the word about the course and helps point the community towards helpful methods to learn PowerShell.

The author sincerely thanks you for purchasing and reading *Mastering Windows PowerShell Scripting*.

Happy coding!

Bibliography

This Learning Path is a blend of content, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Getting Started with PowerShell, Michael Shepard*
- *Windows PowerShell for .Net Developers - Second Edition, Chendrayan Venkatesan, Sherif Talaat*
- *Mastering Windows PowerShell Scripting, Brenton J.W. Blawat*

Index

A

- access control lists
 - copying / [Copying access control lists](#)
- ACL rules
 - adding / [Adding and removing ACL rules](#)
 - removing / [Adding and removing ACL rules](#)
- actions / [Using WMI objects](#)
- advanced functions
 - versus compiled cmdlets / [Getting started with PowerShell scripting](#)
- advanced modules
 - building / [Building advanced scripts and modules](#)
- advanced scripts
 - building / [Building advanced scripts and modules](#)
- aliases
 - using / [Understanding aliases](#)
- anchors, regular expression
 - about / [Regular expression anchors](#)
 - ^ / [Regular expression anchors](#)
 - \A / [Regular expression anchors](#)
 - \$ / [Regular expression anchors](#)
 - \Z / [Regular expression anchors](#)
 - \b / [Regular expression anchors](#)
 - \B / [Regular expression anchors](#)
- AND comparison operator
 - about / [And / OR comparison operators](#)
- API
 - exploring, PowerShell used / [Exploring API using PowerShell](#)
 - Exchange Web Services (EWS) API, for managing Exchange Online / [The EWS API for managing Exchange Online](#)
 - items, purging in mailbox folder / [Purging items in the mailbox folder](#)
 - items, deleting from mailbox folder / [Deleting items from the](#)

[mailbox folder](#)

- Application Lifecycle Management (ALM) / [Using the Azure REST API in PowerShell](#)
- application pools
 - working with / [Working with application pools](#)
 - creating / [Creating application pools](#)
- arrays
 - about / [Arrays](#)
 - single dimension arrays / [Single-dimension arrays](#)
 - jagged arrays / [Jagged arrays](#)
 - values, updating / [Updating array values](#)
- assembly line
 - pipeline / [The pipeline as an assembly line](#)
 - executing / [The pipeline as an assembly line](#)
 - DOS pipeline / [This isn't your DOS or Linux pipeline](#)
 - Linux pipeline / [This isn't your DOS or Linux pipeline](#)
 - objects / [Objects at your disposal](#)
 - summarizing / [Putting them together](#)
- attributes
 - retrieving / [Retrieving attributes and properties](#)
- authoring phase
 - about / [Authoring phase](#)
- Authoring phase, Desired State Configuration / [The Authoring phase](#)
- automatic formatting
 - rules / [The rules of automatic formatting](#)
- Automation
 - exploring / [Exploring COM and Automation](#)
- automation
 - programs, invoking / [Invoking programs for automation](#)
- Azure Resource Manager (ARM) / [Using the Azure REST API in PowerShell](#)

B

- best practices
 - for script management / [Best practices for script management](#)
 - for script creation / [Best practices for script creation](#)

- for software automation / [Best practices for software automation](#)
- best practices, functions / [Best practices for functions](#)
- best practices, looping structures / [Best practices for looping structures and switches](#)
- best practices, script creation
 - about / [Best practices for script creation](#), [Other important best practices for script creation](#)
 - script structure / [Script structure](#)
 - source files, controlling / [Controlling source files](#)
- best practices, switches / [Best practices for looping structures and switches](#)
- binary modules
 - about / [Binary modules](#)
- breakpoints
 - managing / [Managing breakpoints](#)
 - line breakpoint / [Line breakpoints](#)
 - variable breakpoint / [Variable breakpoints](#)
 - command breakpoint / [Command breakpoints](#)
- bytes
 - formatting / [Formatting bytes](#)

C

- # commenting headers
 - about / [# commenting headers](#)
 - comment block / [# commenting headers](#)
 - comment block, starting / [# commenting headers](#)
 - .SYNOPSIS / [# commenting headers](#)
 - .DESCRIPTION / [# commenting headers](#)
 - .PARAMETER / [# commenting headers](#)
 - .EXAMPLE / [# commenting headers](#)
 - .NOTES / [# commenting headers](#)
 - .LINK / [# commenting headers](#)
 - comment block, ending / [# commenting headers](#)
 - code, commenting / [Commenting code](#)
- .create() method / [Managing local users](#)

- CDXML modules
 - about / [CDXML modules](#)
 - advantages / [CDXML modules](#)
- certification authority (CA) / [HTTP/HTTPS Listener](#)
- characters, regular expressions
 - \ / [Getting started with regular expressions](#)
 - \d / [Getting started with regular expressions](#)
 - \D / [Getting started with regular expressions](#)
 - \s / [Getting started with regular expressions](#)
 - \S / [Getting started with regular expressions](#)
 - \w / [Getting started with regular expressions](#)
 - \W / [Getting started with regular expressions](#)
 - . / [Getting started with regular expressions](#)
- characters, regular expressions quantifiers
 - * / [Regular expression quantifiers](#)
 - + / [Regular expression quantifiers](#)
 - ? / [Regular expression quantifiers](#)
 - { } / [Regular expression quantifiers](#)
- CIM
 - about / [WMI and CIM, Exploring Windows Remote Management and CIM](#)
 - and WMI / [WMI and CIM](#)
 - basics / [Basics of WMI and CIM](#)
 - commands, exploring / [Exploring CIM commands](#)
 - methods, exploring / [Exploring CIM methods](#)
 - used, for querying remote machines / [Querying the remote machines using CIM](#)
- CIM cmdlets
 - about / [The CIM cmdlets](#)
 - benefits / [Basics of WMI and CIM](#)
- CIM Object Manager (CIMOM) / [WMI structure](#)
- CIM sessions
 - utilizing / [Utilizing CIM sessions](#)
 - creating / [Creating a session](#)
 - creating, with session options / [Creating a session with session options](#)

- Protocol parameter / [Creating a session with session options](#)
- ProxyAuthentication parameter / [Creating a session with session options](#)
- ProxyCredential parameter / [Creating a session with session options](#)
- UseSSL parameter / [Creating a session with session options](#)
- NoEncryption parameter / [Creating a session with session options](#)
- using, for remote management / [Using sessions for remote management](#)
 - removing / [Removing sessions](#)
- class
 - about / [Types, classes, and objects](#)
- class methods, WMI
 - invoking / [Invoking WMI class methods](#)
- client-side object model (CSOM)
 - about / [Client-side object model – SharePoint Online](#)
 - working / [How does CSOM Work?](#)
 - list, creating / [Creating and deleting list](#)
 - list, deleting / [Creating and deleting list](#)
 - PowerShell modules, creating with SDKs / [Making PowerShell modules with SDKs](#)
- CLIXML
 - about / [CLIXML – a special type of XML](#)
- CLIXML file
 - reading / [CLIXML – a special type of XML](#)
- cmdlet definition XML (CDXML)
 - about / [Working with XML and COM](#)
- cmdlets
 - used, for formatting control / [Format-Table and Format-List](#)
 - about / [Scripting the cmdlet style, Windows PowerShell cmdlets](#)
 - exploring / [Windows PowerShell cmdlets](#)
 - commands / [Windows PowerShell cmdlets](#)
 - retrieving / [Windows PowerShell cmdlets](#)
 - commands, exploring / [Windows PowerShell cmdlets](#)
- cmdlet style

- scripting / [Scripting the cmdlet style](#)
- cmdlet syntax
 - interpreting / [Interpreting the cmdlet syntax](#)
- CodePlex
 - URL / [Client-side object model – SharePoint Online](#)
- code testing, methodologies
 - about / [Methodologies for testing code](#)
 - WhatIf argument, testing / [Testing the –WhatIf argument](#)
 - frequency, testing / [Testing the frequency](#)
 - containers, hit testing / [Hit testing containers](#)
 - production testing, avoiding / [Don't test in production](#)
- Collaborative Application Markup Language (CAML) / [Client-side object model – SharePoint Online](#)
- COM
 - working with / [Working with XML and COM](#)
 - exploring / [Exploring COM and Automation](#)
- command breakpoint
 - about / [Command breakpoints](#)
- commands
 - searching, with Get-Command / [What can you do?](#)
 - searching, with tab completion / [The scripter's secret weapon – tab completion](#)
 - packaging / [Packaging commands](#)
 - execution policy / [Execution policy](#)
 - scripts, types / [Types of scripts](#)
 - scope / [Scopes and scripts](#)
- comma separated values (CSV) / [Replacing and splitting strings](#)
- comment-based Help
 - about / [Comment-based help](#)
- Common Language Runtime (CLR) / [Error and exception handling – parameters](#)
- comparison operators
 - basics / [Comparison operator basics](#)
 - not equal comparison operator / [Equal and not equal comparison](#)
 - equal comparison operator / [Equal and not equal comparison](#)

- less than comparison operator / [Greater than and less than comparison](#)
- greater than comparison operator / [Greater than and less than comparison](#)
- match operator / [Contains, like, and match operators](#)
- contains operator / [Contains, like, and match operators](#)
- like operator / [Contains, like, and match operators](#)
- OR comparison operator / [And / OR comparison operators](#)
- And comparison operator / [And / OR comparison operators](#)
- best practices / [Best practices for comparison operators](#)
- compiled cmdlets
 - versus advanced functions / [Getting started with PowerShell scripting](#)
- configurations scripts
 - creating / [Creating configuration scripts](#)
 - creating, with MOF / [Creating a configuration with MOF](#)
 - Class-Defined DSC resource, creating / [Creating a Class-defined DSC resource](#)
- console host / [PowerShell hosts](#)
- constructors
 - about / [Constructors](#)
- container
 - about / [Another kind of container](#)
- containers
 - selecting, for scripts / [Deciding the best container for your scripts](#)
- contains operator
 - about / [Contains, like, and match operators](#)
- contributions
 - URL / [The EWS API for managing Exchange Online](#)
- Count method
 - about / [Counting and trimming strings](#)
- count method / [Hit testing containers](#)
- CSV
 - objects, importing from / [Import from CSV for quick objects](#)
- CSV file

- process, running to / [Exporting a running process to a CSV file](#)
- CSV files
 - working with / [Working with CSV files](#)
 - objects, outputting to / [Output to CSV for quick reports](#)
- custom DSC resource
 - URL / [Creating a Class-defined DSC resource](#)

D

- data types
 - forcing / [Forcing data types](#)
- date manipulation
 - about / [Date and time manipulation](#)
- date time formatting values
 - reference link / [Date and time manipulation](#)
- declarative programming
 - versus imperative programming / [Imperative versus declarative programming](#)
- default values
 - supplying, for parameters / [Default values for parameters](#)
- deployment modes
 - types / [Types of deployment modes](#)
 - about / [Types of deployment modes](#)
 - push mode / [The push mode](#)
 - pull mode / [The pull mode](#)
- Deployment Workbench / [Invoking programs for automation](#)
- Desired Configuration Management / [Invoking programs for automation](#)
- Desired State Configuration
 - basics / [The basics of Desired State Configuration](#)
 - Authoring phase / [The Authoring phase](#)
 - Staging phase / [The Staging phase](#)
 - \$\$Make it so\$\$ phase / [The "Make it so" phase](#)
- Desired State Configuration (DSC) / [Installing/upgrading PowerShell](#)
 - about / [Introducing Windows PowerShell, Installing Windows Management Framework 5.0](#)
- Distributed COM (DCOM)

- about / [Basics of WMI and CIM](#)
- Distributed Component Object Model (DCOM) / [Utilizing CIM sessions](#)
- Distributed Management Task Force (DMTF)
 - about / [Basics of WMI and CIM](#)
 - / [Using WMI objects](#)
- Do/Until looping structure
 - creating / [Looping structures](#)
 - format / [Looping structures](#)
- Do/While looping structure
 - creating / [Looping structures](#)
 - format / [Looping structures](#)
- Document Object Model (DOM) / [XML file structure](#)
- DOS DIR command
 - about / [The DOS DIR command](#)
- DOS output
 - versus PowerShell output / [PowerShell for comparison](#)
- dot-source / [Scopes and scripts](#)
- drifting configurations
 - detecting / [Detecting and restoring drifting configurations](#)
 - restoring / [Detecting and restoring drifting configurations](#)
- drivers (DLL) / [WMI structure](#)
- DSC
 - prerequisites / [Prerequisites](#)
 - URL / [Installing the WMF 5.0 April 2015 preview](#)
 - about / [Getting started with DSC](#)
 - use cases / [Getting started with DSC](#)
 - architecture / [Getting started with DSC](#)
 - authoring phase / [The Authoring phase, Using desired state configuration](#)
 - staging phase / [The Staging phase, Using desired state configuration](#)
 - make it so phase / [The "Make it so" phase](#)
 - Local Configuration Manager (LCM) / [Local Configuration Manager](#)
 - configuration script, parameterizing / [Parameterizing the](#)

configuration script

- MOF / [Understanding MOF](#)
- using / [Using desired state configuration](#)
- remediation phase / [Using desired state configuration](#)
- archive resource / [Using desired state configuration](#)
- environment resource / [Using desired state configuration](#)
- file resource / [Using desired state configuration](#)
- group resource / [Using desired state configuration](#)
- log resource / [Using desired state configuration](#)
- package resource / [Using desired state configuration](#)
- WindowsProcess resource / [Using desired state configuration](#)
- registry resource / [Using desired state configuration](#)
- WindowsFeature resource / [Using desired state configuration](#)
- script resource / [Using desired state configuration](#)
- service resource / [Using desired state configuration](#)
- user resource / [Using desired state configuration](#)
- resources, URL / [Using desired state configuration](#)
- dynamic module
 - about / [Dynamic modules](#)

E

- encrypting and decrypting strings
 - URL / [Other important best practices for script creation](#)
- equal comparison operator / [Equal and not equal comparison](#)
- error and exception handling
 - parameters / [Error and exception handling – parameters](#)
 - Try / Catch block / [Error and exception handling – Try/Catch](#)
 - legacy exception handling / [Error and exception handling – legacy exception handling](#)
- events
 - about / [What are members?](#)
- EWS Managed API 2.2
 - URL / [The EWS API for managing Exchange Online](#)
- examples, regular expressions / [Regular expressions examples](#)
- Exchange Online
 - managing, Exchange Web Services (EWS) API used / [The EWS](#)

[API for managing Exchange Online](#)

- Exchange Online session
 - connecting to / [Working with XML and COM](#)
- Exchange Web Services (EWS) API
 - for, managing Exchange Online / [The EWS API for managing Exchange Online](#)
- execution policy / [Execution policy](#)
- Export-CSV cmdlet / [Output to CSV for quick reports](#)
- Export-ModuleMember cmdlet
 - about / [The Export-ModuleMember cmdlet](#)
- Extended Type System / [Where did these all come from?](#)
- Extension
 - about / [PowerShell for comparison](#)

F

- FileInfo class
 - reference link / [The Get-Member cmdlet](#)
- files
 - creating, with PowerShell / [Creating files, folders, and registry items with PowerShell](#)
 - verifying / [Verifying files, folders, and registry items](#)
 - moving / [Copying and moving files and folders](#)
 - copying / [Copying and moving files and folders](#)
 - renaming / [Renaming files, folders, registry keys, and named values](#)
 - deleting / [Deleting files, folders, registry keys, and named values](#)
 - extended attributes, viewing / [Viewing file and folder extended attributes](#)
 - mode, setting / [Setting the mode and extended file and folder attributes](#)
 - extended attributes, setting / [Setting the mode and extended file and folder attributes](#)
 - permissions, managing / [Managing file, folder, and registry permissions](#)
- file structure, XML
 - XML parser / [XML file structure](#)

- about / [XML file structure](#)
- folders
 - creating, with PowerShell / [Creating files, folders, and registry items with PowerShell](#)
 - verifying / [Verifying files, folders, and registry items](#)
 - copying / [Copying and moving files and folders](#)
 - moving / [Copying and moving files and folders](#)
 - renaming / [Renaming files, folders, registry keys, and named values](#)
 - deleting / [Deleting files, folders, registry keys, and named values](#)
 - extended attributes, viewing / [Viewing file and folder extended attributes](#)
 - mode, setting / [Setting the mode and extended file and folder attributes](#)
 - extended attributes, setting / [Setting the mode and extended file and folder attributes](#)
- ForEach loop structure
 - about / [Looping structures](#)
 - format / [Looping structures](#)
- For looping structure
 - format / [Looping structures](#)
- Format-Custom cmdlet
 - about / [PowerShell formatting](#)
- Format-List cmdlet / [The Get-Member cmdlet](#)
- Format-Table cmdlet
 - about / [PowerShell formatting](#)
- formatting
 - occurring / [When does formatting occur?](#)
 - controlling, by cmdlets / [Cmdlets that control formatting](#)
 - drawbacks / [The dangers of formatting](#)
 - best practices / [Best practices of formatting](#)
- functions
 - versus scripts / [Comparing scripts and functions](#)
 - executing / [Executing and calling functions](#)
 - calling / [Executing and calling functions](#)
 - packaging / [Packaging functions](#)

- about / [Functions](#)
- declaring / [Functions](#)
- reference link, for advanced parameters / [Functions](#)
- best practices / [Best practices for functions](#)

G

- .GetElementsByTagName() method / [Reading XML files](#)
- Get-Alias cmdlet
 - about / [Understanding aliases](#)
- Get-Command cmdlet
 - used, for finding commands / [What can you do?](#)
 - about / [What can you do?, Understanding aliases](#)
- Get-Content cmdlet / [Reading and writing text files](#)
- Get-FormatData cmdlet
 - about / [PowerShell formatting](#)
- Get-Help
 - used, for working with cmdlets / [How does that work?](#)
- Get-Help cmdlet
 - about / [Getting help](#)
- Get-Member cmdlet
 - about / [The Get-Member cmdlet, Understanding objects](#)
- Get-Service / [The scripter's secret weapon – tab completion](#)
- Get-Service cmdlet / [The Get-Member cmdlet](#)
 - about / [Understanding pipelines](#)
- Get-WMIObject
 - objects, retrieving with / [Retrieving objects with Get-WMIObject](#)
- GetType()
 - URL / [Querying for local users and groups](#)
- GitHub / [Installing the WMF 5.0 April 2015 preview](#)
- GlobalWeather web service / [Using web services](#)
- Graphical User Interface (GUI)
 - about / [Setting up the console host using PowerShell](#)
- greater than comparison operator
 - about / [Greater than and less than comparison](#)
- Group-Object cmdlet

- about / [The Group-Object cmdlet](#)
- grouping constructs, regular expression
 - about / [Regular expression grouping constructs and ranges](#)
 - using, examples / [Regular expression grouping constructs and ranges](#)
- groups
 - managing / [Managing local users and groups](#)
 - querying for / [Querying for local users and groups](#)
- GUI
 - used, for setting up console host / [Setting up the console host using GUI](#)
 - used, for setting up Windows PowerShell ISE host / [Exploring the Windows PowerShell ISE host using GUI](#)

H

- hashes
 - about / [Hashes](#)
- help
 - obtaining / [Getting help](#)
- here-string / [Extending the .NET Framework types](#)

I

- ICANN format / [Regular expressions examples](#)
- IIS
 - installing / [Installing IIS](#)
 - detecting, in Windows 8.1 / [Detecting and installing IIS in Windows 8.1](#)
 - installing, in Windows 8.1 / [Detecting and installing IIS in Windows 8.1](#)
 - installing, in Server 2012R2 / [Detecting and installing IIS in Server 2012R2](#)
 - detecting, in Server 2012R2 / [Detecting and installing IIS in Server 2012R2](#)
 - verifying / [Verifying IIS](#)
 - starting / [Starting, stopping, and restarting IIS](#)
 - stopping / [Starting, stopping, and restarting IIS](#)

- restarting / [Starting, stopping, and restarting IIS](#)
- imperative programming
 - versus declarative programming / [Imperative versus declarative programming](#)
- installed PowerShell version
 - determining / [Determining the installed PowerShell version](#)
 - Registry, used for searching installed version / [Using the registry to find the installed version](#)
 - PowerShell, used for searching installed version / [Using PowerShell to find the installed version](#)
- Integrated Development Environment (IDE) / [PowerShell hosts](#)
- Integrated Scripting Environment (ISE) / [PowerShell hosts](#)
- interactive shell
 - using / [Using an interactive shell](#)
 - cmdlets, using / [Windows PowerShell cmdlets](#)
 - help, obtaining / [Getting help](#)
 - aliases, using / [Understanding aliases](#)
 - regular expressions (regex) / [Understanding expressions](#)
 - objects / [Understanding objects](#)
 - pipelines / [Understanding pipelines](#)
 - filtering / [Understanding filtering and formatting, PowerShell formatting](#)
 - formatting / [Understanding filtering and formatting, PowerShell formatting](#)
 - snippets, exploring in PowerShell ISE / [Exploring snippets in the PowerShell ISE](#)
- Internet Corporation for Assigned Names and Numbers (ICANN) / [Regular expressions examples](#)
- Invoke-Item cmdlet
 - about / [The Invoke-Item cmdlet](#)
- InvokeMember()
 - URL / [Querying for local users and groups](#)
- IPCONFIG command
 - about / [The IPCONFIG command](#)

- jagged arrays
 - about / [Jagged arrays](#)
- JavaScript object model (JSOM) / [Client-side object model – SharePoint Online](#)
- JavaScript Object Notation (JSON)
 - exploring / [Exploring JSON](#)

L

- legacy exception handling / [Error and exception handling – legacy exception handling](#)
- Length method
 - about / [Counting and trimming strings](#)
- Length property / [The Where-Object cmdlet](#)
- less than comparison operator
 - about / [Greater than and less than comparison](#)
- like operator
 - about / [Contains, like, and match operators](#)
- line-of-business (LOB) / [The Lync 2013 client-side API](#)
- line breakpoint
 - about / [Line breakpoints](#)
- Local Configuration Manager (LCM)
 - about / [The "Make it so" phase](#)
 - / [The "Make it so" phase, Local Configuration Manager](#)
- local groups
 - managing / [Managing local groups](#)
- local users
 - managing / [Managing local users and groups](#), [Managing local users](#)
 - querying for / [Querying for local users and groups](#)
- logic
 - adding / [Adding some logic](#)
 - conditional logic (IF) / [Conditional logic \(IF\)](#)
 - looping / [Looping logic](#)
 - while / [More logic](#)
 - Do..Until / [More logic](#)
 - For / [More logic](#)

- looping structures
 - about / [Looping structures](#)
 - Do/While / [Looping structures](#)
 - Do/Until / [Looping structures](#)
 - ForEach / [Looping structures](#)
 - For / [Looping structures](#)
 - best practices / [Best practices for looping structures and switches](#)
- Lync 2013
 - about / [The Lync 2013 client-side API](#)
 - client settings, exploring / [Exploring client settings](#)
 - test calls, automating / [Automating test calls](#)
 - IM, with contacts / [IM with contacts](#)
- Lync Extensibility API
 - URL / [The Lync 2013 client-side API](#)
- Lync Model API
 - URL / [The Lync 2013 client-side API](#)
- LYNC SDK
 - installing / [Installation of LYNC SDK](#)
 - URL / [Installation of LYNC SDK](#)

M

- \$\$Make it so\$\$ phase, Desired State Configuration / [The "Make it so" phase](#)
- -match operator / [Getting started with regular expressions](#)
- Managed Object Format (MOF) / [WMI structure](#)
- Managed Object Format (MOF) file
 - about / [Basics of WMI and CIM](#)
- Management OData Schema Designer
 - URL / [Creating the Management OData web service](#)
- Management OData Web Service
 - URL / [Creating the Management OData web service](#)
- manifest modules
 - about / [Manifest modules, Manifest modules](#)
 - benefits / [Manifest modules](#)
- match operator

- about / [Contains, like, and match operators](#)
- Measure-Command cmdlet
 - about / [Getting started with PowerShell scripting](#)
- Measure-Object cmdlet / [What can you do?](#)
 - about / [The Measure-Object cmdlet](#)
- members
 - about / [Types, classes, and objects](#), [What are members?](#)
- methods
 - about / [What are members?](#)
 - calling / [Calling methods](#)
- Microsoft Connect
 - URL / [Scripting the cmdlet style](#)
- Microsoft Deployment Toolkit (MDT) / [Invoking programs for automation](#)
- MicroSoft Developers Network (MSDN)
 - about / [The Get-Member cmdlet](#)
- Microsoft Online Services Module for Windows PowerShell
 - URL, for downloading / [Working with XML and COM](#)
- Microsoft Online Services Sign-In Assistant
 - URL, for downloading / [Working with XML and COM](#)
- mode attributes
 - d--- / [Retrieving attributes and properties](#)
 - -a--- / [Retrieving attributes and properties](#)
 - ---h- / [Retrieving attributes and properties](#)
 - ----s / [Retrieving attributes and properties](#)
- modules
 - about / [Packaging functions](#), [Introduction to modules](#)
 - location / [Where do modules live?](#)
 - removing / [Removing a module](#), [Removing a module – take two](#)
 - listing / [Listing modules](#)
 - script modules / [Script modules](#)
 - binary modules / [Binary modules](#)
 - manifest modules / [Manifest modules](#)
 - dynamic module / [Dynamic modules](#)
- MOF
 - about / [Understanding MOF](#)

- tasks, performing / [Understanding MOF](#)
- URL / [Understanding MOF](#)
- used, for creating configurations / [Creating a configuration with MOF](#)
/ [Authoring phase](#)
- MSDN
 - URL / [The EWS API for managing Exchange Online](#)

N

- .NET Framework 4
 - URL, for download / [Prerequisites](#)
- .NET Framework types
 - extending / [Extending the .NET Framework types](#)
- .NET objects
 - exploring / [Exploring .NET objects](#)
 - creating / [Creating .NET objects](#)
 - extending, for Administrations and Development tasks / [Extending .NET objects for Administrations and Development tasks](#)
- named values
 - adding, to registry keys / [Adding named values to registry keys](#)
 - renaming / [Renaming files, folders, registry keys, and named values](#)
 - deleting / [Deleting files, folders, registry keys, and named values](#)
- namespace / [WMI structure](#)
- naming conventions
 - about / [Naming conventions](#)
- New-Variable cmdlet
 - about / [Getting started with PowerShell scripting](#)
- New Technology File System (NTFS) / [Viewing file and folder extended attributes](#)
- not equal comparison operator / [Equal and not equal comparison](#)
- number manipulation
 - about / [Number manipulation and parsing](#)
- numbers
 - formatting / [Formatting numbers](#)

O

- objects
 - about / [Objects all the way down, Types, classes, and objects](#)
 - outputting, to CSV files / [Output to CSV for quick reports](#)
 - importing, from CSV / [Import from CSV for quick objects](#)
 - retrieving, with Get-WMIOBJECT / [Retrieving objects with Get-WMIOBJECT](#)
 - using / [Understanding objects](#)
 - stored, in variables / [Objects stored in variables](#)
- OCSPowerShell
 - URL / [Exploring WSMAN cmdlets](#)
- Open Data Protocol (OData)
 - about / [Management OData IIS Extensions](#)
 - IIS Extensions, managing / [Management OData IIS Extensions](#)
 - Management OData web service, creating / [Creating the Management OData web service](#)
- Operating System (OS) / [Using the registry to find the installed version](#)
- OR comparison operator
 - about / [And / OR comparison operators](#)
- out-file cmdlet
 - about / [The out-file cmdlet](#)

P

- Package Management
 - exploring / [Exploring Package Management](#)
- parameters
 - used, for adding flexibility / [Parameters add flexibility](#)
 - about / [Parameters add flexibility, Parameters revisited](#)
 - typed parameters / [Typed parameters](#)
 - switch type / [Switches](#)
 - default values, supplying for / [Default values for parameters](#)
 - WarningAction / [Error and exception handling – parameters](#)
 - ErrorAction / [Error and exception handling – parameters](#)
 - ErrorVariable / [Error and exception handling – parameters](#)

- WarningVariable / [Error and exception handling – parameters](#)
 - Try / Catch block, using with / [Error and exception handling – Try/Catch with parameters](#)
- ParameterSets / [Interpreting the cmdlet syntax](#)
- parsing
 - about / [Number manipulation and parsing](#)
- pipe / [The Get-Member cmdlet](#)
 - about / [Piping variables](#)
- pipeline
 - as assembly line / [The pipeline as an assembly line](#)
- pipeline data
 - dealing with / [Dealing with pipeline data](#)
 - Sort-Object cmdlet / [The Sort-Object cmdlet](#)
 - Where-Object cmdlet / [The Where-Object cmdlet](#)
 - Select-Object cmdlet / [The Select-Object cmdlet](#)
 - Measure-Object cmdlet / [The Measure-Object cmdlet](#)
 - Group-Object cmdlet / [The Group-Object cmdlet](#)
- pipelines
 - using / [Understanding pipelines](#)
 - process, running to CSV file / [Exporting a running process to a CSV file](#)
- piping
 - about / [Piping variables](#)
 - variables / [Piping variables](#)
- PLOTPROJECTILE function
 - about / [Digging into objects](#)
- PowerShell
 - used, for finding installed PowerShell version / [Using PowerShell to find the installed version](#)
 - installing / [Installing/upgrading PowerShell](#)
 - upgrading / [Installing/upgrading PowerShell](#)
 - versions, URLs / [Installing/upgrading PowerShell](#)
 - session, starting / [Starting a PowerShell session](#)
 - host / [PowerShell hosts](#)
 - 64-bit / [64-bit and 32-bit PowerShell](#)
 - 32-bit / [64-bit and 32-bit PowerShell](#)

- as administrator / [PowerShell as an administrator](#)
- commands / [Simple PowerShell commands](#)
- aliases / [PowerShell aliases](#)
- pipeline / [The pipeline as an assembly line](#)
- used, for setting up console host / [Setting up the console host using PowerShell](#)
- used, for parsing structured objects / [Parsing structured objects using PowerShell](#)
- used, for exploring API / [Exploring API using PowerShell](#)
- comparison operators / [Comparison operator basics](#)
- PowerShell alias / [PowerShell aliases](#)
- PowerShell Gallery
 - URL / [Exploring PowerShellGet](#)
- PowerShellGet
 - exploring / [Exploring PowerShellGet](#)
- PowerShell ISE
 - snippets, exploring in / [Exploring snippets in the PowerShell ISE](#)
- PowerShell module
 - autoloading / [PowerShell module autoloading](#)
- PowerShell modules
 - about / [Understanding PowerShell modules](#)
 - creating, with SDKs / [Making PowerShell modules with SDKs](#)
 - URL / [Best practices for software automation](#)
- PowerShell output
 - versus DOS output / [PowerShell for comparison](#)
- PowerShell redirection
 - about / [PowerShell streams and redirection](#)
- PowerShell session
 - starting / [Starting a PowerShell session](#)
- PowerShell streams
 - about / [PowerShell streams and redirection](#)
- PowerShell Web Access (PSWA)
 - about / [PowerShell Web Access](#)
 - installing / [Installing PowerShell Web Access](#)
 - configuring / [Configuring PowerShell Web Access](#)
 - authorization rules, applying / [Applying authorization rules](#)

- prerequisites, DSC / [Prerequisites](#)
- profiles
 - about / [Profiles](#)
- properties
 - about / [What are members?](#)
 - retrieving / [Retrieving attributes and properties](#)
- PSDrives
 - about / [Comparing scripts and functions](#)
- PSProviders
 - about / [Comparing scripts and functions](#)
- pull mode, deployment modes
 - about / [The pull mode](#)
 - pull server, creating with SMB Share / [Creating a pull server using the SMB share](#)
 - pull server, creating with HTTP and HTTP(s) / [Creating the pull server using HTTP and HTTPS](#)
- push mode, deployment modes / [The push mode](#)

R

- #Requires statement / [The #Requires statement](#)
- ranges, regular expression
 - about / [Regular expression grouping constructs and ranges](#)
- redirection operators
 - about / [Other types of redirection operators](#)
- Registry
 - used, for finding installed PowerShell version / [Using the registry to find the installed version](#)
- registry items
 - creating, with PowerShell / [Creating files, folders, and registry items with PowerShell](#)
 - verifying / [Verifying files, folders, and registry items](#)
- registry keys
 - named values, adding / [Adding named values to registry keys](#)
 - renaming / [Renaming files, folders, registry keys, and named values](#)
- registry permissions

- managing / [Managing file, folder, and registry permissions](#)
- registry provider / [Registry provider](#)
- regular expression quantifiers
 - about / [Regular expression quantifiers](#)
 - characters / [Regular expression quantifiers](#)
- regular expressions
 - about / [Getting started with regular expressions](#)
 - grouping construct / [Regular expression grouping constructs and ranges](#)
 - ranges / [Regular expression grouping constructs and ranges](#)
 - quantifiers / [Regular expression quantifiers](#)
 - anchors / [Regular expression anchors](#)
 - examples / [Regular expressions examples](#)
- regular expressions (regex)
 - using / [Understanding expressions](#)
 - reference link / [Understanding expressions](#)
- remediation phase
 - about / [Staging and remediation phase](#)
- Remove-Variable cmdlet
 - about / [Getting started with PowerShell scripting](#)
- Replace() method
 - used, for replacing strings / [Replacing and splitting strings](#)
- report issues
 - URL / [The EWS API for managing Exchange Online](#)
- Representation State Transfer (REST) / [Exploring web requests](#)
- request batching
 - URL / [How does CSOM Work?](#)
- REST API
 - exploring / [Exploring the REST API, Using the Azure REST API in PowerShell](#)
- return keyword
 - about / [Output](#)
- rules, automatic formatting
 - about / [The rules of automatic formatting](#)
 - files, formatting / [Formatting files](#)
 - decisions, based on first object / [Formatting decisions are based](#)

on the first object

- small objects, inserting in table / [Small objects go in a table](#)
- large objects, inserting in list / [Large objects go in a list](#)

S

- scope
 - creating, for execution / [Scopes and scripts](#)
- scriptblock syntax / [The Where-Object cmdlet](#)
- Script Browser
 - about / [The PowerShell ISE script browser](#)
 - options / [The PowerShell ISE script browser](#)
 - benefits / [The PowerShell ISE script browser](#)
 - URL / [The PowerShell ISE script browser](#)
- script creation
 - best practices / [Best practices for script creation](#)
- script debugging
 - about / [Script debugging, Debugging scripts](#)
- scripting
 - in cmdlet style / [Getting started with PowerShell scripting](#)
 - principles / [Getting started with PowerShell scripting](#)
 - variables, using / [Getting started with PowerShell scripting](#)
 - using / [Getting started with PowerShell scripting](#)
 - consideration / [Getting started with PowerShell scripting](#)
 - creating / [Getting started with PowerShell scripting](#)
 - executing / [Getting started with PowerShell scripting](#)
 - advanced functions, writing / [Getting started with PowerShell scripting](#)
 - compiled cmdlet, creating / [Getting started with PowerShell scripting](#)
 - advanced functions, using / [Getting started with PowerShell scripting](#)
- script management
 - best practices / [Best practices for script management](#)
- script modules
 - about / [Script modules, Script modules](#)
- scripts

- about / [Another kind of container](#)
 - versus functions / [Comparing scripts and functions](#)
 - container, selecting / [Deciding the best container for your scripts](#)
- scripts, types
 - controller scripts / [Types of scripts](#)
 - tool scripts / [Types of scripts](#)
- Security Account Manager (SAM) / [Creating property instances, Managing local users](#)
- Select-Object cmdlet
 - using, ways / [The Select-Object cmdlet](#)
 - objects returned count, limiting / [Limiting the number of objects returned](#)
 - objects returned properties, limiting / [Limiting the properties of the objects returned](#)
 - single property values, retrieving / [Retrieving the values of a single property](#)
 - about / [Understanding filtering and formatting](#)
- Server 2012R2
 - IIS, installing in / [Detecting and installing IIS in Server 2012R2](#)
 - IIS, detecting in / [Detecting and installing IIS in Server 2012R2](#)
- Server Message Block (SMB) / [The Staging phase](#)
- ServiceController class
 - URL / [Windows PowerShell cmdlets](#)
- serviceExample function / [Error and exception handling – parameters](#)
- SharePoint 2010 CSOM
 - URL / [Client-side object model – SharePoint Online](#)
- SharePoint 2013 CSOM
 - URL / [Client-side object model – SharePoint Online](#)
- SharePoint Online CSOM
 - URL / [Client-side object model – SharePoint Online](#)
- Simple Object Access Protocol (SOAP) / [Exploring web services](#)
- single dimension arrays
 - about / [Single-dimension arrays](#)
- Software Inventory Logging (SIL)

- about / [Exploring Package Management](#)
- Sort-Object cmdlet
 - about / [The Sort-Object cmdlet](#)
- Split() method
 - used, for splitting strings / [Replacing and splitting strings](#)
- staging phase
 - about / [Staging and remediation phase](#)
- Staging phase, Desired State Configuration / [The Staging phase](#)
- static fields
 - about / [Number manipulation and parsing](#)
- Stop-Service cmdlet
 - about / [Understanding pipelines](#)
- string manipulation
 - about / [String manipulation](#)
- strings
 - replacing, Replace() method used / [Replacing and splitting strings](#)
 - splitting, Split() method used / [Replacing and splitting strings](#)
 - counting / [Counting and trimming strings](#)
 - trimming / [Counting and trimming strings, The Trim method](#)
 - true method / [The string true and false methods](#)
 - false method / [The string true and false methods](#)
- structured objects
 - parsing, PowerShell used / [Parsing structured objects using PowerShell](#)
- structures
 - usage, combining of / [Combining the use of functions, switches, and loops](#)
- Substring() method
 - about / [The Substring method](#)
- Switch command / [Switches](#)
- switches
 - about / [Switches, Switches](#)
 - best practices / [Best practices for looping structures and switches](#)
- System.Math class

- URL / [Using an interactive shell](#)
- System Center Configuration Manager (SCCM) / [WMI structure](#), [Invoking programs for automation](#), [Staging and remediation phase](#)
- System Center Orchestrator / [Invoking programs for automation](#)

T

- tab completion
 - used, for finding commands / [The scripter's secret weapon – tab completion](#)
 - working / [How does that work?](#)
- TechNet Gallery
 - URL / [Installing the WMF 5.0 April 2015 preview](#)
- text
 - outputting / [Output](#)
- text files
 - reading / [Reading and writing text files](#)
 - writing / [Reading and writing text files](#), [Writing text files](#)
- time manipulation
 - about / [Date and time manipulation](#)
- toLower() method / [String manipulation](#)
- toUpper() method / [String manipulation](#)
- Trim() method
 - about / [The Trim method](#)
- TrimEnd() method / [The Trim method](#)
- TrimStart() method / [The Trim method](#)
- Try / Catch block
 - using / [Error and exception handling – Try/Catch](#)
 - using, with parameters / [Error and exception handling – Try/Catch with parameters](#)
- type
 - about / [Types, classes, and objects](#)
- typed parameters
 - about / [Typed parameters](#)

U

- Uniform Resource Identifier (URI) / [Exploring web requests](#)

- Universal Characters Set Transformation Format 8-bit/16-bit (UTF-8/16) / [XML file structure](#)
- Universal Description, Discovery, and Integration (UDDI) / [Exploring web services](#)
- Updatable Help feature
 - about / [Manifest modules](#)
- Update-Help cmdlet
 - about / [Getting help](#)
- use case of classes, WMF 5.0
 - about / [Use case of classes in WMF 5.0](#)
 - constructors / [Constructors](#)
- User Account Control (UAC) / [PowerShell as an administrator](#), [Creating files, folders, and registry items with PowerShell](#)

V

- variable breakpoint
 - about / [Variable breakpoints](#)
- variables
 - using / [Getting started with PowerShell scripting](#)
 - about / [Variables](#)
 - objects, stored / [Objects stored in variables](#)
 - piping / [Piping variables](#)
- virtual directories
 - creating / [Creating virtual directories and web applications](#)
- Visual Studio
 - URL / [How does CSOM Work?](#)

W

- WebAdministration module
 - about / [The WebAdministration module](#)
- web applications
 - creating / [Creating virtual directories and web applications](#)
- web requests
 - exploring / [Exploring web requests](#)
 - files, downloading from Internet / [Downloading files from the Internet](#)

- files, reading from Internet / [Reading a file from the Internet](#)
- web services
 - exploring / [Exploring web services](#)
 - using / [Using web services](#)
 - URL / [Using web services](#)
 - building / [Building web services](#)
- Web Services Description Language (WSDL) / [Exploring web services](#)
- Web Services for Management (WS-MAN)
 - about / [WMI and CIM](#)
- Web Services Management (WS-Man)
 - abilities / [Windows PowerShell remoting](#)
 - cmdlets, exploring / [Exploring WSMAN cmdlets](#)
 - cmdlets, URL / [Exploring WSMAN cmdlets](#)
- WhatIf argument
 - testing / [Testing the –WhatIf argument](#)
- Where-Object cmdlet
 - about / [The Where-Object cmdlet, Understanding filtering and formatting](#)
- Windows 8.1
 - IIS, installing in / [Detecting and installing IIS in Windows 8.1](#)
 - IIS, detecting in / [Detecting and installing IIS in Windows 8.1](#)
- Windows Data Protection API
 - about / [Working with XML and COM](#)
- Windows features
 - installing / [Installing Windows features and roles](#)
- Windows Management Framework (WMF) / [Installing/upgrading PowerShell](#)
- Windows Management Framework 5.0
 - installing / [Installing Windows Management Framework 5.0](#)
- Windows Management Instrumentation (WMI) / [Installing/upgrading PowerShell, Exploring CIM commands](#)
- Windows PowerShell
 - about / [Introducing Windows PowerShell](#)
 - URL / [Introducing Windows PowerShell](#)
- Windows PowerShell 5.0

- reference link / [Scripting the cmdlet style](#)
 - about / [Exploring Windows PowerShell 5.0](#)
 - reference link, for benefits / [Exploring Windows PowerShell 5.0](#)
 - highlights / [Exploring Windows PowerShell 5.0](#)
 - basics, of Desired State Configuration / [The basics of Desired State Configuration](#)
 - Package Management, exploring / [Exploring Package Management](#)
 - PowerShellGet, exploring / [Exploring PowerShellGet](#)
- Windows PowerShell console host
 - about / [The Windows PowerShell consoles, The Windows PowerShell console host](#)
 - setting up, GUI used / [Setting up the console host using GUI](#)
 - setting up, PowerShell used / [Setting up the console host using PowerShell](#)
- Windows PowerShell consoles
 - console host / [The Windows PowerShell consoles](#)
 - Windows PowerShell ISE host / [The Windows PowerShell consoles](#)
- Windows PowerShell ISE host
 - about / [The Windows PowerShell consoles](#)
 - exploring, GUI used / [Exploring the Windows PowerShell ISE host using GUI](#)
 - benefits / [Benefits of an ISE](#)
 - Script Browser / [The PowerShell ISE script browser](#)
- Windows processes
 - managing / [Managing Windows processes](#)
- Windows Remoting (WinRM) / [Installing/upgrading PowerShell](#)
- Windows roles
 - installing / [Installing Windows features and roles](#)
- Windows Server 2012 R2
 - URL / [Installing Windows features and roles](#)
- Windows services
 - managing / [Managing Windows services](#)
- WinRM
 - about / [Exploring Windows Remote Management and CIM,](#)

[Exploring CIM commands](#)

- HTTP/HTTPS Listener / [HTTP/HTTPS Listener](#)
- WMF 4.0
 - URL, for download / [Prerequisites](#)
- WMF 5.0
 - use case of classes / [Use case of classes in WMF 5.0](#)
- WMF 5.0 April 2015 preview
 - installing / [Installing the WMF 5.0 April 2015 preview](#)
 - URL / [Installing the WMF 5.0 April 2015 preview](#), [Creating a Class-defined DSC resource](#)
- WMI
 - about / [What is WMI?](#)
 - and CIM / [WMI and CIM](#)
 - basics / [Basics of WMI and CIM](#)
 - references / [Basics of WMI and CIM](#)
 - structure / [WMI structure](#)
 - class methods, invoking / [Invoking WMI class methods](#)
- WMI classes
 - finding / [Finding WMI classes](#)
 - searching / [Searching for WMI classes](#)
- WMIClass type accelerators
 - [WMI] / [Basics of WMI and CIM](#)
 - [WMIClass] / [Basics of WMI and CIM](#)
 - [WMISearcher] / [Basics of WMI and CIM](#)
- WMI consumers / [WMI structure](#)
- WMI infrastructure / [WMI structure](#)
- WMI objects
 - using / [Using WMI objects](#)
- WMI organization
 - about / [WMI organization](#)
- WMI property instances
 - creating / [Creating, modifying, and removing WMI property instances](#), [Creating property instances](#)
 - modifying / [Creating, modifying, and removing WMI property instances](#), [Modifying property instances](#)
 - removing / [Creating, modifying, and removing WMI property instances](#)

[instances](#), [Removing property instances](#)

- WMI provider / [WMI structure](#)
- WMI Query Language (WQL)
 - about / [Basics of WMI and CIM](#)
 - / [Invoking WMI class methods](#)
- WMI structure
 - WMI infrastructure / [WMI structure](#)
 - WMI providers / [WMI structure](#)
 - WMI consumers / [WMI structure](#)
- WQL
 - about / [Getting the right instance](#)
 - syntax / [WQL syntax](#)
- write-progress cmdlet
 - URL / [Other important best practices for script creation](#)
- WS-Management (WS-Man) / [Exploring CIM commands](#)
- WSMan
 - about / [Basics of WMI and CIM](#)
- WSMan object
 - about / [Basics of WMI and CIM](#)

X

- xHotfix resource / [Installing the WMF 5.0 April 2015 preview](#)
- XML
 - working with / [Working with XML and COM](#)
- XML content
 - adding / [Adding XML content](#)
 - modifying / [Modifying XML content](#)
 - removing / [Removing XML content](#)
- XML files
 - reading / [Reading XML files](#)
- XML parser / [XML file structure](#)

Table of Contents

PowerShell: Automating Administrative Tasks	16
PowerShell: Automating Administrative Tasks	17
Credits	18
Preface	20
What this learning path covers	20
What you need for this learning path	22
Who this learning path is for	23
Reader feedback	24
Customer support	25
Downloading the example code	25
Errata	26
Piracy	26
Questions	28
1. Module 1	29
1. First Steps	30
Determining the installed PowerShell version	30
Using the registry to find the installed version	30
Using PowerShell to find the installed version	32
Installing/upgrading PowerShell	34
Starting a PowerShell session	36
PowerShell hosts	36
64-bit and 32-bit PowerShell	39
PowerShell as an administrator	40
Simple PowerShell commands	43
PowerShell aliases	46
Summary	48
For further reading	49
2. Building Blocks	50
What can you do?	50
The scripter's secret weapon – tab completion	55

How does that work?	56
Interpreting the cmdlet syntax	62
Summary	64
For further reading	65
3. Objects and PowerShell	66
Objects all the way down	66
Digging into objects	66
Types, classes, and objects	67
What are members?	68
The DOS DIR command	68
The IPCONFIG command	69
PowerShell for comparison	70
The Get-Member cmdlet	72
Where did these all come from?	76
Summary	77
For further reading	78
4. Life on the Assembly Line	79
The pipeline as an assembly line	79
This isn't your DOS or Linux pipeline	80
Objects at your disposal	81
Dealing with pipeline data	82
The Sort-Object cmdlet	82
The Where-Object cmdlet	85
The Select-Object cmdlet	87
Limiting the number of objects returned	87
Limiting the properties of the objects returned	88
Retrieving the values of a single property	89
The Measure-Object cmdlet	92
The Group-Object cmdlet	93
Putting them together	97
Summary	98
For further reading	99

5. Formatting Output	100
When does formatting occur?	100
The rules of automatic formatting	102
Formatting files	102
Formatting decisions are based on the first object	103
Small objects go in a table	105
Large objects go in a list	107
Cmdlets that control formatting	109
Format-Table and Format-List	109
The dangers of formatting	113
Best practices of formatting	114
Summary	115
For further reading	116
6. Scripts	117
Packaging commands	117
Execution policy	118
Types of scripts	120
Scopes and scripts	121
Parameters add flexibility	123
Adding some logic	125
Conditional logic (IF)	125
Looping logic	126
More logic	127
Profiles	129
Summary	131
For further reading	132
7. Functions	133
Another kind of container	133
Comparing scripts and functions	133
Executing and calling functions	136
Naming conventions	138
Comment-based help	139

Parameters revisited	143
Typed parameters	143
Switches	145
Default values for parameters	146
Output	147
Summary	150
For further reading	151
8. Modules	152
Packaging functions	152
Script modules	154
The Export-ModuleMember cmdlet	155
Where do modules live?	157
Removing a module	158
PowerShell module autoloading	159
The #Requires statement	160
Removing a module – take two	161
Manifest modules	161
Listing modules	163
Summary	166
9. File I/O	167
Reading and writing text files	167
Writing text files	168
Working with CSV files	170
Output to CSV for quick reports	170
The Invoke-Item cmdlet	171
Import from CSV for quick objects	172
PowerShell streams and redirection	174
Other types of redirection operators	174
The out-file cmdlet	175
CLIXML – a special type of XML	176
Summary	180

For further reading	181
10. WMI and CIM	182
What is WMI?	182
WMI organization	182
Finding WMI classes	184
Retrieving objects with Get-WMIOBJECT	186
Getting the right instance	189
WQL syntax	189
Calling methods	192
WMI and CIM	194
The CIM cmdlets	194
CDXML modules	196
Summary	197
For further reading	198
11. Web Server Administration	199
Installing IIS	199
Detecting and installing IIS in Windows 8.1	199
Detecting and installing IIS in Server 2012R2	200
Verifying IIS	202
The WebAdministration module	203
Starting, stopping, and restarting IIS	205
Creating virtual directories and web applications	207
Working with application pools	210
Creating application pools	212
Summary	214
For further reading	215
A. Next Steps	216
2. Module 2	217
1. Getting Started with Windows PowerShell	218
Scripting the cmdlet style	218
Introducing Windows PowerShell	221

Installing Windows Management Framework 5.0	222
The Windows PowerShell consoles	228
The Windows PowerShell console host	229
Setting up the console host using GUI	231
Setting up the console host using PowerShell	236
Exploring the Windows PowerShell ISE host using GUI	241
Benefits of an ISE	245
The PowerShell ISE script browser	245
Using an interactive shell	249
Windows PowerShell cmdlets	250
Getting help	254
Understanding aliases	258
Understanding expressions	265
Understanding objects	272
Understanding pipelines	276
Exporting a running process to a CSV file	278
Understanding filtering and formatting	282
PowerShell formatting	289
Exploring snippets in the PowerShell ISE	294
Getting started with PowerShell scripting	297
Summary	315
2. Unleashing Development Skills Using Windows PowerShell 5.0	316
Basics of WMI and CIM	316
Working with XML and COM	330
Exploring COM and Automation	337
Exploring .NET objects	350
Creating .NET objects	350
Extending .NET objects for Administrations and Development tasks	352
Extending the .NET Framework types	354
Building advanced scripts and modules	358
Exploring Windows PowerShell 5.0	372

The basics of Desired State Configuration	372
The Authoring phase	377
The Staging phase	377
The "Make it so" phase	377
Use case of classes in WMF 5.0	378
Constructors	383
Parsing structured objects using PowerShell	385
Exploring Package Management	389
Exploring PowerShellGet	393
Understanding PowerShell modules	396
Introduction to modules	396
Script modules	397
Binary modules	398
Manifest modules	403
Dynamic modules	404
Script debugging	406
Managing breakpoints	406
Line breakpoints	407
Variable breakpoints	407
Command breakpoints	407
Debugging scripts	408
Summary	411
3. Exploring Desired State Configuration	412
Prerequisites	413
Installing the WMF 5.0 April 2015 preview	416
Imperative versus declarative programming	420
Getting started with DSC	424
The Authoring phase	425
The Staging phase	427
The "Make it so" phase	427
Local Configuration Manager	428

Parameterizing the configuration script	429
Understanding MOF	430
Exploring Windows Remote Management and CIM	433
Windows PowerShell remoting	433
Exploring WSMAN cmdlets	434
HTTP/HTTPS Listener	439
Exploring CIM commands	442
Exploring CIM methods	446
Querying the remote machines using CIM	448
Creating configuration scripts	450
Creating a configuration with MOF	450
Creating a Class-defined DSC resource	453
Types of deployment modes	460
The push mode	460
The pull mode	461
Creating a pull server using the SMB share	461
Creating the pull server using HTTP and HTTPS	464
Summary	468
4. PowerShell and Web Technologies	469
PowerShell Web Access	469
Installing PowerShell Web Access	469
Configuring PowerShell Web Access	474
Applying authorization rules	475
Management OData IIS Extensions	477
Creating the Management OData web service	480
Exploring web requests	481
Downloading files from the Internet	484
Reading a file from the Internet	485
Exploring web services	488
Using web services	488
Building web services	493

Exploring the REST API	497
Using the Azure REST API in PowerShell	497
Exploring JSON	501
Summary	504
5. Exploring Application Programming Interface	505
Exploring API using PowerShell	510
The EWS API for managing Exchange Online	513
Purging items in the mailbox folder	519
Deleting items from the mailbox folder	521
The Lync 2013 client-side API	522
Installation of LYNC SDK	522
Exploring client settings	526
Automating test calls	531
IM with contacts	533
Client-side object model – SharePoint Online	536
How does CSOM Work?	537
Creating and deleting list	539
Making PowerShell modules with SDKs	541
Summary	546
3. Module 3	547
1. Variables, Arrays, and Hashes	548
Variables	549
Objects stored in variables	551
Arrays	554
Single-dimension arrays	554
Jagged arrays	556
Updating array values	557
Hashes	560
Deciding the best container for your scripts	564
Summary	565
2. Data Parsing and Manipulation	566
String manipulation	566

Replacing and splitting strings	567
Counting and trimming strings	569
The Trim method	570
The Substring method	573
The string true and false methods	574
Number manipulation and parsing	577
Formatting numbers	578
Formatting bytes	580
Date and time manipulation	582
Forcing data types	586
Piping variables	589
Summary	592
3. Comparison Operators	593
Comparison operator basics	593
Equal and not equal comparison	595
Greater than and less than comparison	597
Contains, like, and match operators	599
And / OR comparison operators	603
Best practices for comparison operators	605
Summary	606
4. Functions, Switches, and Loops Structures	607
Functions	607
Looping structures	615
Switches	621
Combining the use of functions, switches, and loops	623
Best practices for functions, switches, and loops	626
Best practices for functions	626
Best practices for looping structures and switches	626
Summary	628
5. Regular Expressions	630
Getting started with regular expressions	631
Regular expression grouping constructs and ranges	637

Regular expression quantifiers	640
Regular expression anchors	647
Regular expressions examples	652
Summary	656
6. Error and Exception Handling and Testing Code	657
Error and exception handling – parameters	658
Error and exception handling – Try/Catch	661
Error and exception handling – Try/Catch with parameters	662
Error and exception handling – legacy exception handling	665
Methodologies for testing code	669
Testing the –WhatIf argument	669
Testing the frequency	670
Hit testing containers	671
Don't test in production	673
Summary	675
7. Session-based Remote Management	677
Utilizing CIM sessions	678
Creating a session	682
Creating a session with session options	684
Using sessions for remote management	686
Removing sessions	688
Summary	691
8. Managing Files, Folders, and Registry Items	692
Registry provider	692
Creating files, folders, and registry items with PowerShell	694
Adding named values to registry keys	698
Verifying files, folders, and registry items	701
Copying and moving files and folders	703
Renaming files, folders, registry keys, and named values	707
Deleting files, folders, registry keys, and named values	712
Summary	717
9. File, Folder, and Registry Attributes, ACLs, and Properties	718

Retrieving attributes and properties	719
Viewing file and folder extended attributes	724
Setting the mode and extended file and folder attributes	727
Managing file, folder, and registry permissions	732
Copying access control lists	732
Adding and removing ACL rules	735
Summary	743
10. Windows Management Instrumentation	745
WMI structure	745
Using WMI objects	748
Searching for WMI classes	751
Creating, modifying, and removing WMI property instances	756
Creating property instances	759
Modifying property instances	760
Removing property instances	762
Invoking WMI class methods	764
Summary	767
11. XML Manipulation	768
XML file structure	768
Reading XML files	772
Adding XML content	777
Modifying XML content	781
Removing XML content	784
Summary	787
12. Managing Microsoft Systems with PowerShell	788
Managing local users and groups	788
Managing local users	789
Managing local groups	791
Querying for local users and groups	797
Managing Windows services	802
Managing Windows processes	809
Installing Windows features and roles	814

Summary	818
13. Automation of the Environment	819
Invoking programs for automation	819
Using desired state configuration	826
Authoring phase	828
Staging and remediation phase	832
Detecting and restoring drifting configurations	835
Summary	839
14. Script Creation Best Practices and Conclusion	840
Best practices for script management	840
# commenting headers	841
Commenting code	843
Best practices for script creation	845
Script structure	845
Other important best practices for script creation	846
Controlling source files	849
Best practices for software automation	850
Summary	852
Mastering Windows PowerShell Scripting – conclusion	852
Staying connected with the author	853
Bibliography	854
Index	855