



Working with Windows PowerShell

Contents at a Glance

CHAPTER 1:	Introducing PowerShell	503
	Understanding the Basics of PowerShell	503
	Using PowerShell	509
	Running PowerShell Remotely	527
	Getting Help in PowerShell	528
	Identifying Security Issues with PowerShell	530
CHAPTER 2:	Understanding the .NET Framework	535
	Introducing the Various Versions of .NET Framework	535
	Focusing on New Features in .NET 4.7	538
	Viewing the Global Assembly Cache	539
	Understanding .NET Standard and .NET Core	541
CHAPTER 3:	Working with Scripts and Cmdlets	543
	Introducing Common Scripts and Cmdlets	543
	Executing Scripts or Cmdlets	544
	Working from Another Location	546
	Performing Simple Administrative Tasks with PowerShell Scripts	548
CHAPTER 4:	Creating Your Own Scripts and Advanced Functions	551
	Creating a PowerShell Script	552
	Defining a Script Policy	558
	Signing a PowerShell Script	559
	Creating a PowerShell Advanced Function	559
CHAPTER 5:	PowerShell Desired State Configuration	567
	Getting an Overview of PowerShell Desired State Configuration	567
	Creating a PowerShell Desired State Configuration Script	572
	Applying the PowerShell Desired State Configuration Script	573
	Push and Pull: Using PowerShell Desired State Configuration at Scale	575

IN THIS CHAPTER

- » Beginning to use PowerShell, components, and concepts
- » Using PowerShell day to day
- » Using PowerShell to remotely administer systems
- » Getting help in PowerShell
- » Addressing security issues with PowerShell

Chapter **1**

Introducing PowerShell

PowerShell is the wave of the future. As server automation becomes more commonplace, there will be a much higher demand for a system administrator who has PowerShell skills. Additionally, remote administration with PowerShell reduces the need to interact with the graphical user interface (GUI) and can allow you to make changes to one or many systems from your workstation.

This chapter covers the basics of PowerShell, from the beginning terminology to using PowerShell remoting.

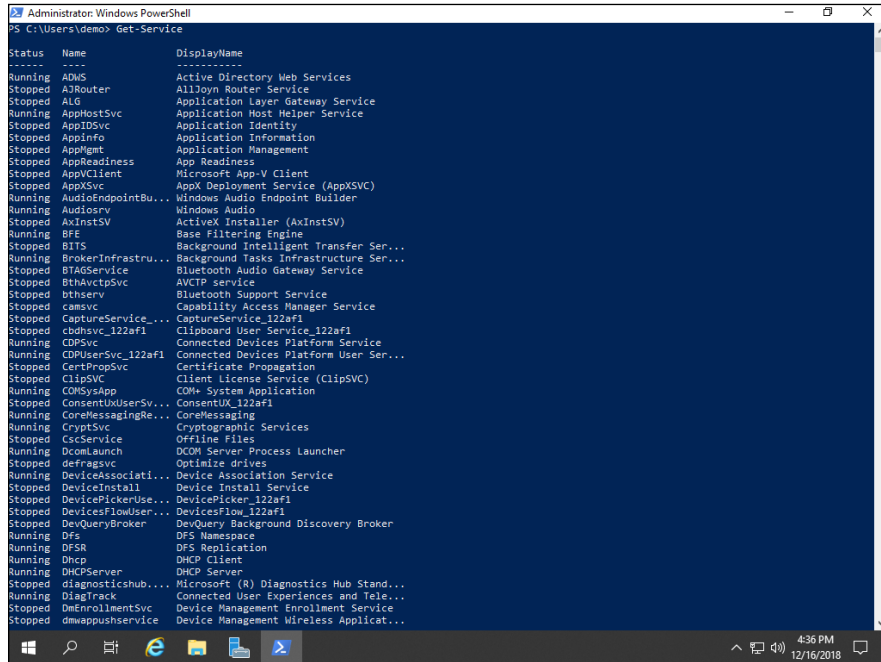
Understanding the Basics of PowerShell

Before you start using a scripting language, you need to learn it. In this section, I fill you in on the basics of PowerShell so the rest of this chapter makes sense.

Objects

In PowerShell, an *object* is a single instance of something, like a service or a process. If you run `Get-Process` or `Get-Service` inside of the PowerShell, each row is an object (assuming you're using the default formatting). In Figure 1-1, there is a service on each line, and each service is an object.

FIGURE 1-1:
Each row
returned in the
`Get-Service`
command is an
object.



Pipeline

The *pipeline* allows you to take the output of one command and send it (pipe it) to the next command. For instance, if you're trying to figure out what methods and properties are available for a cmdlet, you can use the output of the cmdlet and pipe it to `Get-Member`. In Figure 1-2, I've entered in `Get-Service` with no filters, and then added `Get-Member`. In this case, because there are no filters, `Get-Member` is able to return all methods and properties associated with `Get-Service`.

You can do lots of things after the pipeline. Usually, you see things like formatting and filters, sometimes even export commands.

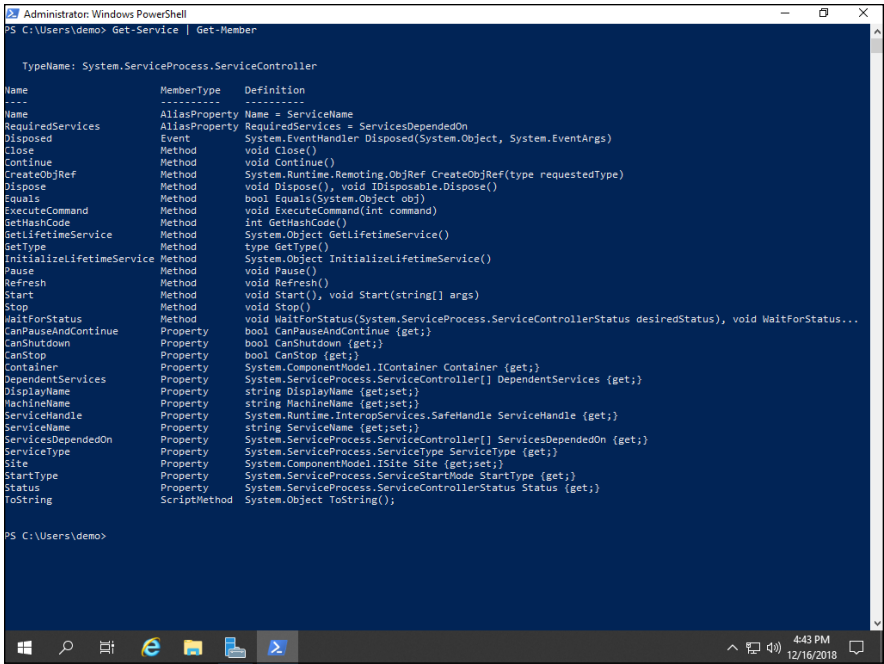


FIGURE 1-2:
Using the pipeline
allows you to
send the output
of one command
to the input of
another.

Providers

PowerShell *providers* allow you to access various data sources as if they were a regular data drive on your system. The providers that are built into PowerShell are listed in Table 1-1. You can view a list of the built-in providers on your system with the following command:

```
Get-PSProvider
```

TABLE 1-1

PowerShell Built-in Providers

Provider	Drive Example
Registry	HKLM:
Alias	Alias:
Environment	Env:
FileSystem	C:
Function	Function:
Variable	Variable:

PowerShell providers allow you to interact with your system in different ways. For instance, the `FileSystem` provider allows you to work with the file system on your server using the `PSDrive` cmdlets. `Get-PSDrive` will return a list of all the drives that are available from within the current PowerShell session, including your system drives, as well as the drives that can be used to work with the other providers. You can also create and remove a `PSDrive` with the `New-PSDrive` and `Remove-PSDrive` cmdlets.

Variables

Think of a *variable* as a container in which you can store something. You can create a variable at any time. Variables are not case-sensitive, and they can use spaces and special characters. As a best practice, Microsoft avoids using spaces and special characters in variable names. It recommends sticking with alphanumeric characters (A–Z, 0–9) and the underscore character (`_`).

Variables are declared with the dollar sign (`$`). For instance, if I wanted to create a variable to store my first name, I could type the following:

```
$FirstName = Sara
```



REMEMBER

Variables are not case-sensitive. I capitalize the first letter of each word because it makes it easier for me to read it, but you don't have to.

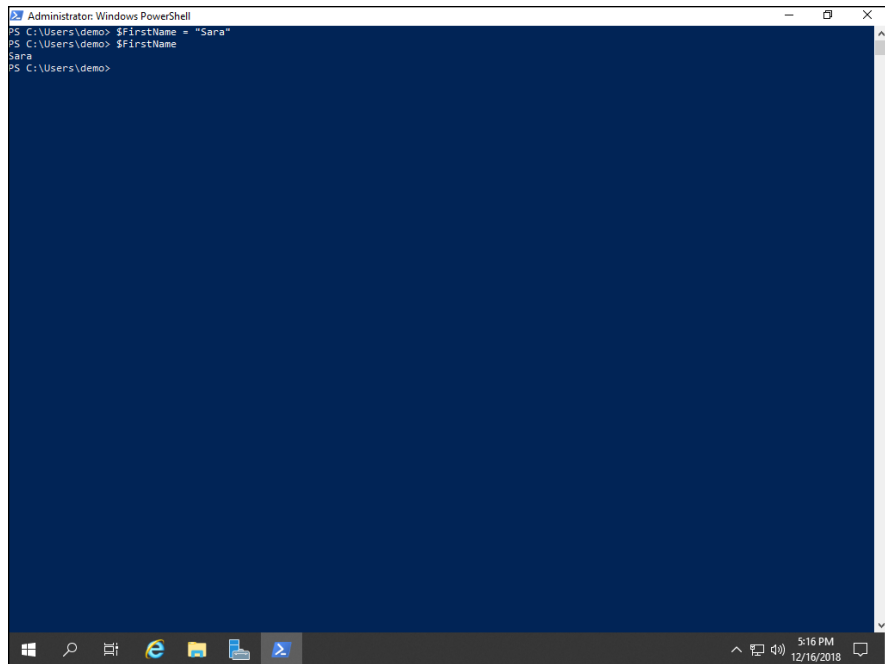
When you want to display the value of a variable, all you need to do is type it with the preceding dollar sign. In Figure 1-3, you can see that I set my variable `FirstName` to my first name `Sara`. Then I typed in the name of the variable and it showed me the value of the variable.

Sessions

Sessions are used to connect to PowerShell on a remote system. This can be done to run commands or to interact directly with the PowerShell on the remote system.

For example, `New-PSSession` will create a new PowerShell session. This can be done on a local system or a remote system. Note that `New-PSSession` creates a persistent connection, whereas `Enter-PSSession` connects to a remote system but only creates a temporary connection that exists for as long as the command or commands are running.

FIGURE 1-3:
Using the dollar sign tells the system you're working with a variable. You can create and display variables by adding the dollar sign in front of the variable name.



Comments

Whenever you write PowerShell scripts, you should always add *comments*, which help others understand how the code works and what it's doing. Comments can also help you if you ever need to change the code, or if you don't run it very often. Comments start with the hash symbol (#). Everything that comes after the hash symbol on that line is a part of the comment. See the following example:

```
#This is a comment...  
Write-Host "This is not a comment"
```

Aliases

Aliases are shortcuts for full commands. There are far too many to list all of the PowerShell aliases, but Table 1-2 lists some of the more common ones.

Cmdlets

A *cmdlet* is a piece of code that consists of a verb and a noun. Common verbs are `Get`, `Set`, `New`, `Install`, and so on. With the cmdlet `Get-Command`, `Get` is the verb and `Command` is the noun.

TABLE 1-2

Common PowerShell Aliases

Alias	Full Command
<code>gcm</code>	<code>Get-Command</code>
<code>sort</code>	<code>Sort-Object</code>
<code>gi</code>	<code>Get-Item</code>
<code>cp</code>	<code>Copy-Item</code>
<code>fl</code>	<code>Format-List</code>
<code>ft</code>	<code>Format-Table</code>
<code>pwd</code>	<code>Get-Location</code>
<code>cls</code>	<code>Clear-Host</code>
<code>ni</code>	<code>New-Item</code>
<code>sleep</code>	<code>Start-Sleep</code>
<code>write</code>	<code>Write-Output</code>
<code>where</code>	<code>Where-Object</code>

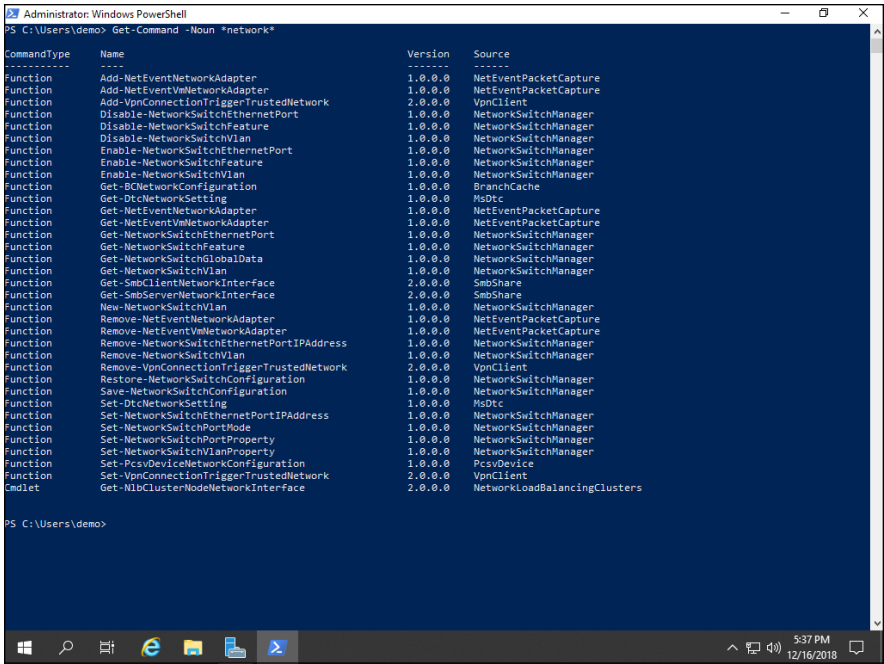
Running `Get-Command` returns all cmdlets, aliases, and functions. You can find cmdlets, aliases, and functions that you're looking for with the `Get-Command` cmdlet. Look at the following:

```
Get-Command -Noun *network*
```

When you run this command, PowerShell returns a list of all commands where the noun includes *network*. The asterisks are wildcards and basically tell PowerShell that you don't care if there is text before or after the noun you're looking for. An example of this command is shown in Figure 1-4.

Parameters allow you to refine what you're interested in the command doing. For instance, using the `Get-Command` cmdlet by default will return cmdlets, aliases, and functions. By using the `-All` parameter, you can get the `Get-Command` cmdlet to return cmdlets, aliases, functions, filters, scripts, and applications. You can find out which parameters are available for a given command by checking the help documentation for the command. I cover using help within PowerShell later in this chapter.

FIGURE 1-4:
You can use the cmdlet `Get-Command` to find other commands even if you don't know the whole noun's name.



Using PowerShell

In this section, I show you the typical day-to-day usage of PowerShell and how to accomplish the things that will make your life as a system administrator so much better.

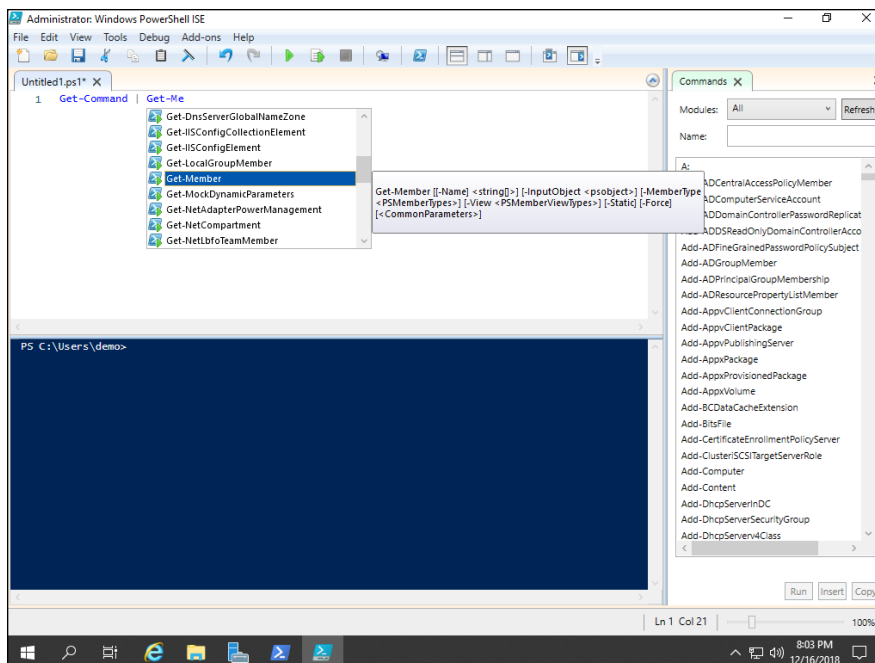
Writing PowerShell commands and scripts

You may end up spending a significant amount of time typing out one-liners in PowerShell, or you may end up writing whole scripts. Several tools work well for writing PowerShell. I cover them in the following sections.

PowerShell Integrated Scripting Environment

You can always type directly into PowerShell, however if you're newer to PowerShell, I recommend using the PowerShell Integrated Scripting Environment (ISE). It makes suggestions based on what you're typing and is very handy if you aren't sure. As you can see in Figure 1-5, PowerShell ISE is correctly suggesting `Get-Member` based on what I started typing. Plus, you can look up commands in the right-side pane. PowerShell ISE is being deprecated; it still exists in PowerShell v5, but it will be removed in PowerShell v6.

FIGURE 1-5:
The PowerShell
ISE is a very
powerful tool
for writing
PowerShell
scripts.



Text editors

For some system administrators, the simplicity of a simple text editor can be really tempting. Notepad is available by default in the Windows Server operating system. Personally, I love Notepad ++ because it's a simple interface, but it still provides color coding and, with the installation of an extension, you can check the differences between two files. Notepad ++ is an open-source project, and you do have to download and install it (go to <https://notepad-plus-plus.org>).

Visual Studio Code

Visual Studio Code is a code editor that is designed to be a light version of the traditional and more complex products. It's optimized for quick code development and available on Windows, Linux, and macOS, free of charge. It features many of the useful features that you're used to if you've used PowerShell ISE in the past, but it adds some new features that make it a better organizational tool. It can complete commands with IntelliSense and has a much more user-friendly method of browsing PowerShell commands when you install the PowerShell extension.

For system administrators who have some familiarity with Visual Studio, or who want integration with GitHub or other code repositories, Visual Studio Code is a great option. It brings support for Git commands out of the box and can be customized with other extensions past the core PowerShell extension. Going forward, it's what Microsoft recommends using for work in PowerShell, given that the PowerShell ISE is being deprecated.

You can download Visual Studio Code at <https://code.visualstudio.com>. Installation and configuration are relatively simple:

- 1. Download the Visual Studio Code installer from** <https://code.visualstudio.com>.
- 2. Navigate to your Downloads folder and double-click the VS Code executable to begin installation.**
- 3. On the Setup – Visual Studio Code screen, click Next.**
- 4. On the License Agreement screen, select the I Accept the Agreement radio button, and then click Next.**
- 5. On the Select Destination Location screen, select where you want the software to install to, and then click Next.**
- 6. On the Select Start Menu Folder screen, choose a folder in the Start menu if you want the application to be stored somewhere other than the default, and click Next.**

You can also opt to not create a Start menu folder. I'll accept the defaults.
- 7. On the Select Additional Tasks screen, select the Create a Desktop Icon check box, the Add "Open with Code" Action to Windows Explorer File Context Menu check box, and the Register Code as an Editor for Supported File Types check box; leave the Add to PATH check box selected (see Figure 1-6).**
- 8. On the Ready to Install screen, click Install.**
- 9. On the Completing the Visual Studio Code Setup Wizard screen, leave Launch Visual Studio Code check box selected, and click Finish.**
- 10. After VS Code has launched, click Tools and Languages.**
- 11. Type PowerShell into the search box, and click Install on the extension for PowerShell, as shown in Figure 1-7.**

FIGURE 1-6:
The Select Additional Tasks screen allows you to customize how and when you'll interact with VS Code.

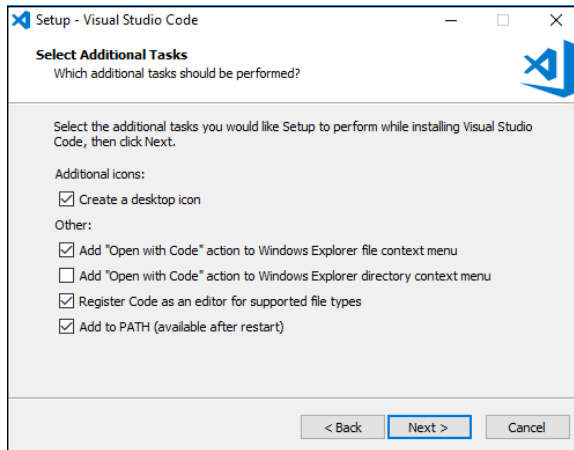
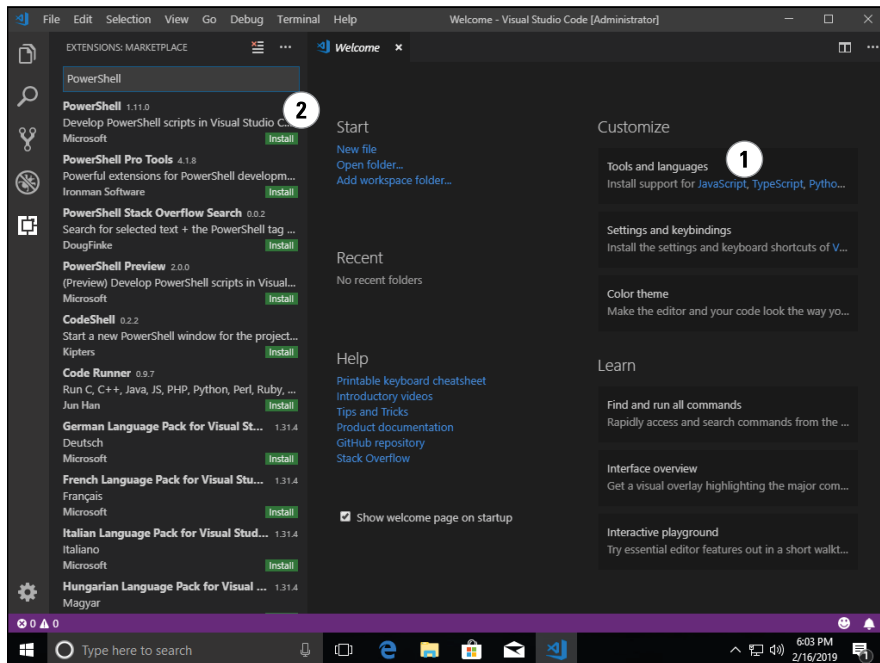
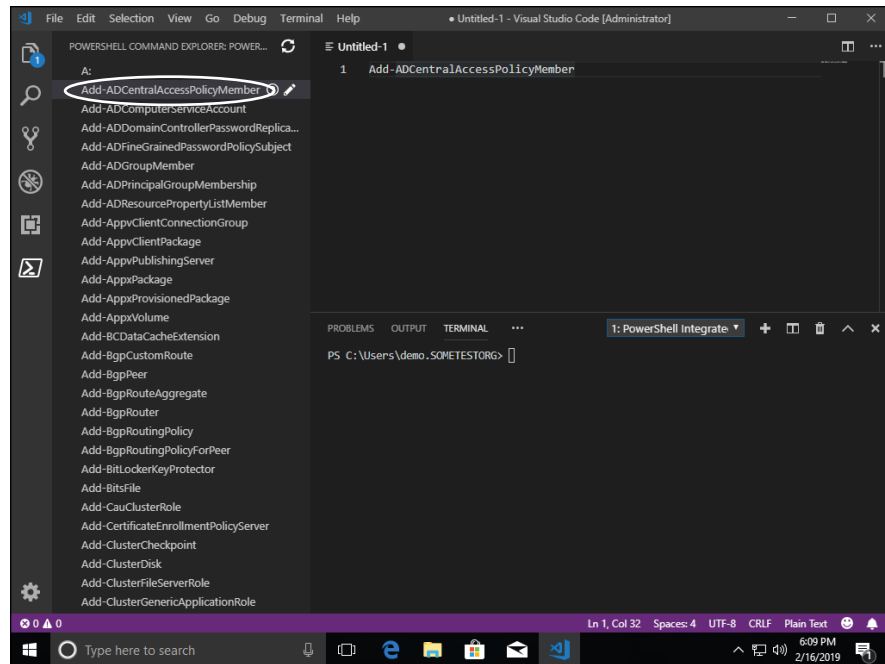


FIGURE 1-7:
After VS Code is installed, you can install the PowerShell extension from the Extension Marketplace.



With the PowerShell extension installed in VS Code, you can open the PowerShell Command Explorer (see Figure 1-8). This is helpful when looking up commands. If you click the question mark next to a cmdlet, your browser opens to the online help page for that cmdlet. Clicking the pencil icon inserts the cmdlet into the coding window.

FIGURE 1-8:
The PowerShell
Command
Explorer windows
shows you
PowerShell
cmdlets and gives
easy access to
help files.



Working with objects

One of the most common things you do in PowerShell is work with objects. In fact, it would be extremely difficult not to! In the following sections, I cover some of the more common objects you use in PowerShell, and some examples of how to work with them.

Properties and methods

Properties and methods are used along with cmdlets to refine what you want the cmdlet to do. Properties are used to view the data that applies to an object, such as checking to see if an object is read-only, or to verify that data regarding an object exists. Properties are prefixed with a dash like this:

- » `Get-Command -version`: Returns the version numbers on everything returned by the `Get-Command` cmdlet
- » `Get-Command -verb Get`: Returns aliases, cmdlets, and functions that use the verb `Get`

Methods are different from properties. You call them by putting a period (.) before the method name. Methods are typically used to specify some kind of action you want to take on an object. Consider the `Replace` method in the following example:

```
'This is a great For Dummies book!'.Replace('great','super')
```

Here, you're using the `Replace` method to change the word *great* to the word *super*.

Variables

As I mention earlier, variables are used to store values. They can store commands, values, and strings. You'll use variables extensively if you do any amount of scripting because they make things simpler. You can call a variable instead of having to type a long command, for example.

Arrays

An *array* is a type of variable. When you create the array variable, you assign multiple values to the same variable. For instance:

```
$Alphabet = A,B,C,D,E,F
```

This code creates an array variable named `$alphabet`, which contains the letters *A, B, C, D, E, and F*.

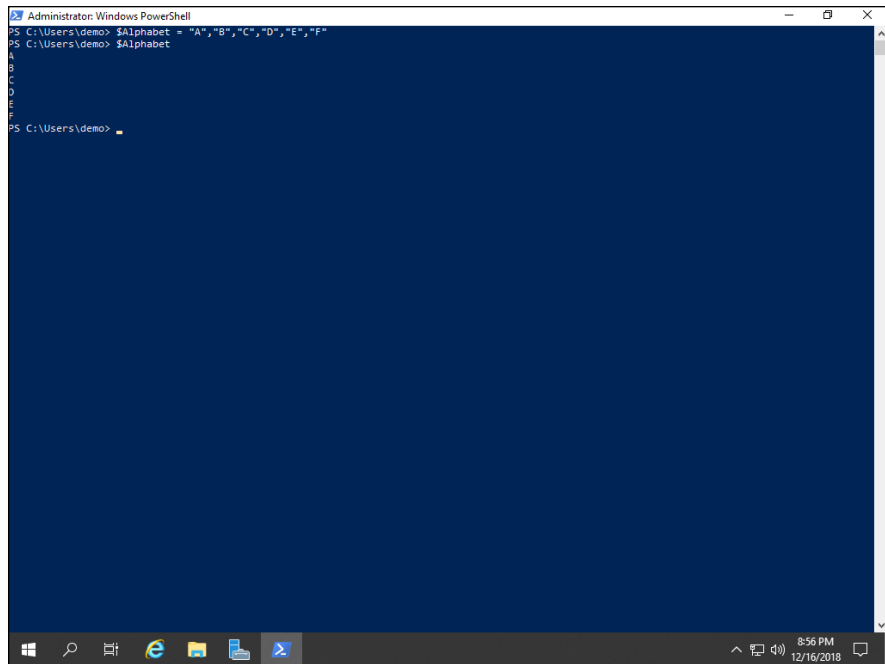
To read back the contents of the array, you do the same thing that you would do to display the contents of a variable. You call the variable in the PowerShell window — in this case, `$Alphabet`. You get a display similar to Figure 1-9 if you use my example.

Arrays are a very useful type of variable for system administration work. For example, when you want to store a list of usernames or computer names that you exported from Active Directory so that you can run commands against each of them, an array is a perfect fit. I haven't covered loops yet, but you can run a command against each entry in an array with a loop, instead of having to run a command against an individual system.

Working with the pipeline

A *pipeline* is essentially a group of commands that are connected with a pipe to form a pipeline. The first command sends its output to the input of the second command, and the second command sends its output to the input of the third command.

FIGURE 1-9:
Creating an array
variable and
displaying an
array variable
are very similar
to what you used
earlier to work
with variables.



The pipeline can be very handy for passing commands and/or data to another command so that you can act on it. Say, for instance, the calculator is running and you hate that pesky calculator. You can run the following command to end it.

```
Get-Process win32calc | Stop-Process
```

Sure you could've hit the red X to close the calculator, but what fun would that be? In the real world, you would use this command to stop a process that isn't responding. You could use a similar version with `Start-Process` or other useful cmdlets, and run them against one or more servers to simplify remote administration tasks.



TIP

The `$_` symbol represents whatever objects happens to be in the pipeline at that point in time. If you need to filter on two or more properties, this represents a really nice shorthand method to do this. Consider the following example:

```
Get-Process | Where {$_ .CPU -gt 50}
Get-Process | Where {$_ .CPU -gt 50 -AND $_ .Handles -gt 1000}
```

In this case, the `$_` is passing the output of the `Get-Process` cmdlet and is allowing you to access the properties of the `Get-Process` command to run the comparison operators against. You can see the actual output of these two commands in Figure 1-10.

```

PS C:\Users\demo> Get-Process | Where ($_.CPU -gt 30)

Handles NPM(K) PM(K) WS(K) CPU(s) Id SI ProcessName
-----
489 64 120504 78376 119.92 3528 0 MsMpEng
412 14 16168 20320 66.83 1464 0 svchost
307 15 16948 18620 38.69 5588 0 svchost
2068 0 204 872 314.67 4 0 System
435 27 14468 33552 32.30 3116 1 vmtoolsd

PS C:\Users\demo> Get-Process | Where ($_.CPU -gt 30 -AND $_.Handles -lt 1500)

Handles NPM(K) PM(K) WS(K) CPU(s) Id SI ProcessName
-----
489 64 120504 78376 119.92 3528 0 MsMpEng
412 14 16168 20320 66.84 1464 0 svchost
306 15 16948 18620 38.69 5588 0 svchost
435 27 14468 33552 32.30 3116 1 vmtoolsd

PS C:\Users\demo>

```

FIGURE 1-10: Using a pipeline with `$_` to pass on the output of the first cmdlet, which allows you to use the cmdlet's properties to filter using the comparison operators.



TIP

So, when should you use the pipeline? I believe that you should use the pipeline anytime you need the output of one command to be passed to the next command. This can result in some very long pipelines, but using the pipeline is simpler than breaking each cmdlet out on its own.

Working with modules

PowerShell is very powerful out of the box, but you'll run into scenarios when it can't accomplish what you want it to do. For example, PowerShell out of the box does not know how to work with Active Directory objects. To tell it how to interact with Active Directory, you import the `ActiveDirectory` module. At its simplest, a module is just a package that contains cmdlets, providers, variables, and functions.

Browsing available modules

Modules are pretty cool, right? Your next question may be: How do I find out what modules are available to me? I'm glad you asked. The command is simple:

```
Get-Module -ListAvailable
```

If you run the command by itself, you'll get a list of the modules that are available currently within your session. By adding the `-ListAvailable` parameter, you can get a list of modules that are installed on your computer and available for use. This command will take some time to return results, but when it does, you'll have a nice list of all the PowerShell modules that are available to you on your system. I've included an example of the output in Figure 1-11. Please note that your output may look different because you'll have additional modules listed based on the roles and features installed on your system.

```

Select Administrator: Windows PowerShell
PS C:\Users\demo> Get-Module -ListAvailable

Directory: C:\Program Files\WindowsPowerShell\Modules
-----
ModuleType Version      Name                               ExportedCommands
-----
Script      1.0.1      Microsoft.PowerShell.Operation.V... {Get-OperationValidation, Invoke-OperationValidation}
Binary      1.0.0.1    PackageManagement                  {Find-Package, Get-Package, Get-PackageProvider, Get-PackageSource...}
Script      3.4.0      Pester                             {Describe, Context, It, Should...}
Script      1.0.0.1    PowerShellGet                      {Install-Module, Find-Module, Save-Module, Update-Module...}
Script      2.0.0      PSReadline                        {Get-PSReadlineKeyHandler, Set-PSReadlineKeyHandler, Remove-PSReadlineKeyHandler,...}

Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules
-----
ModuleType Version      Name                               ExportedCommands
-----
Manifest 1.0.1.0      ActiveDirectory                   {Add-ADCentralAccessPolicyMember, Add-ADComputerServiceAccount, Add-ADDomainContr...
Manifest 1.0.0.0      ADOSDeployment                    {Add-ADOSReadOnlyDomainControllerAccount, Install-ADOSForest, Install-ADOSDomain...
Manifest 1.0.0.0      AppBackgroundTask                {Disable-AppBackgroundTaskDiagnosticLog, Enable-AppBackgroundTaskDiagnosticLog, 5...
Manifest 2.0.0.0      AppLocker                        {Add-AppLockerFileInformation, Get-AppLockerPolicy, New-AppLockerPolicy, Set-Appl...
Manifest 1.0.0.0      AppvClient                       {Add-AppvClientConnectionGroup, Add-AppvClientPackage, Add-AppvPublishingServer, ...
Manifest 2.0.1.0      Appx                             {Add-AppxPackage, Get-AppxPackage, Get-AppxPackageManifest, Remove-AppxPackage...}
Manifest 1.0      BestPractices                    {Get-BpaModel, Get-BpaResult, Invoke-BpaModel, Set-BpaResult}
Manifest 2.0.0.0      BitsTransfer                     {Add-BitsFile, Complete-BitsTransfer, Get-BitsTransfer, Remove-BitsTransfer...}
Manifest 1.0.0.0      BranchCache                     {Add-BCDataCacheExtension, Clear-BCCache, Disable-BC, Disable-BCDowngrading...}
Manifest 1.0.0.0      CimCmdlets                      {Get-CimAssociatedInstance, Get-CimClass, Get-CimInstance, Get-CimSession...}
Manifest 1.0      ConfigCI                        {Get-SystemDriver, New-CIPolicyRule, New-CIPolicy, Get-CIPolicy...}
Manifest 1.0      Defender                        {Get-MpPreference, Set-MpPreference, Add-MpPreference, Remove-MpPreference...}
Manifest 1.0.1.0      DeliveryOptimization             {Get-DeliveryOptimizationStatus, Get-DeliveryOptimizationPerfSnap, Get-DeliveryOp...
Manifest 1.0      Dfsn                            {Get-DfsnRoot, Remove-DfsnRoot, Set-DfsnRoot, New-DfsnRoot...}
Manifest 2.0.0.0      DhcpServer                      {Add-DhcpServerInDC, Add-DhcpServerv4Class, Add-DhcpServerv4ExclusionRange, Add-D...
Manifest 1.0.0.0      DirectAccessClientComponents     {Disable-DAManualEntryPointSelection, Enable-DAManualEntryPointSelection, Get-DAC...
Script      3.0      Dism                             {Add-AppxProvisionedPackage, Add-WindowsDriver, Add-WindowsCapability, Add-Window...
Manifest 1.0.0.0      DnsClient                       {Resolve-DnsName, Clear-DnsClientCache, Get-DnsClient, Get-DnsClientCache...}
Manifest 2.0.0.0      DnsServer                      {Add-DnsServerConditionalForwarderZone, Add-DnsServerDirectoryPartition, Add-DnsS...
Manifest 1.0.0.0      EventTracingManagement          {Start-EtwTraceSession, New-EtwTraceSession, Get-EtwTraceSession, Update-EtwTrace...
Manifest 1.0.0.0      GroupPolicy                     {Backup-GPO, Block-GPInheritance, Copy-GPO, Get-GPInheritance...}
Script      1.1.0.0      IISAdministration              {Get-IISAppPool, Start-IISCommitDelay, Stop-IISCommitDelay, Get-IISSite...}
Manifest 2.0.0.0      International                   {Get-WinDefaultInputMethodOverride, Set-WinDefaultInputMethodOverride, Get-WinHom...
Manifest 1.0.0.0      IScsi                           {Get-IscliTargetPortal, New-IscliTargetPortal, Remove-IscliTargetPortal, Update-I...
Manifest 2.0.0.0      IScsiTarget                    {Add-ClusterISCSITargetServerRole, Add-IscliVirtualDiskTargetMapping, Checkpoint-...
Script      1.0.0.0      Ise                             {New-IseSnippet, Import-IseSnippet, Get-IseSnippet}
Manifest 1.0.0.0      Kds                             {Add-KdsRootKey, Get-KdsRootKey, Test-KdsRootKey, Set-KdsConfiguration...}
Manifest 1.0.1.0      Microsoft.PowerShell.Archive    {Compress-Archive, Expand-Archive}
Manifest 3.0.0.0      Microsoft.PowerShell.Diagnostics {Get-WinEvent, Get-Counter, Import-Counter, Export-Counter...}
Manifest 3.0.0.0      Microsoft.PowerShell.Host       {Start-Transcript, Stop-Transcript}

```

FIGURE 1-11: Viewing the available modules in PowerShell gives you an idea of how powerful it can be, and how many tools you have at your disposal.

Browsing the properties of a module

`Get-Member` is the simplest method to find out what properties are available to you with modules. The following line of code will give you a printout of all the properties associated with `Get-Module`. The last part of the command cuts the results

down to just the name column because you aren't interested in the other columns for this purpose.

```
Get-Module | Get-Member -MemberType Property | Format-Table Name
```

Working with comparison operators

Comparison operators are very useful when you **need to see if two objects match or don't match**. For instance, you may use a comparison operator to find all the services on your system that are disabled. In this section, I cover the most common comparison operators.

-eq and -ne

Equal to (-eq) and not equal to (-ne) are used when you need to find an exact match to something, or when you want to ensure your results do not match. The response you get back is a true/false response. Consider the following code sample:

```
$num = 2
$othernum = 3
$num -eq $othernum
```

When the last line runs, it will return a false because 2 does not equal 3. Let's run not equal to (-ne) now, which returns a true:

```
$num -ne $othernum
```

You can see these little snippets in Figure 1-12 with their outputs.

-gt and -lt

Greater than (-gt) and less than (-lt) are also comparison operators that return a true/false response. You can play with a really simple version of this by setting a variable to a value and then testing it. For example:

```
$x = 4
$x -gt 8
```

The preceding code will check to see if the value of x (in this case 4) is greater than 8. It is not, so the response will be false.

FIGURE 1-12:
If you're trying to determine whether one object matches another, a simple equal to (–eq) may be your best bet.

```

Administrator: Windows PowerShell
PS C:\Users\demo> $num=2
PS C:\Users\demo> $othernum=3
PS C:\Users\demo> $num -eq $othernum
False
PS C:\Users\demo> $num -ne $othernum
True
PS C:\Users\demo>
  
```

-and and -or

–and and –or also return true/false responses based on the conditions that they're given. –and, for example, returns a true if both statements it is fed are true. –or returns a true if one of the statements it is given is true.

For example, the following statement will be true if \$a is less than \$b, and \$b is less than 50:

```
($a -lt $b) -and ($b -lt 50)
```

This equation is similar but will be true if \$a is less than \$b, or if \$b is less than 50:

```
($a -lt $b) -or ($b -lt 50)
```

Getting information out of PowerShell

It's all fun and games to write equations and have PowerShell put the answer right on the screen, but the truth of the matter is that you're usually going to want PowerShell to output the information in a different format. A common use case is exporting information on systems out of Active Directory to a CSV, so that you can filter on properties of each system like the operating system version, and whether

service packs are installed. Or you may want it to write the result on the screen, but with some kind of text to give the information context. In the following sections, I examine a few of the ways to get PowerShell to output text.

Write-Host

`Write-Host` is used to write things onscreen. This command is very helpful when you want to give the result of something some context.

For example, you can set the value of a variable, in this case, `$x` to 2:

```
$x = 2
```

Then you can use `Write-Host` to print a sentence and the output of the variable:

```
Write-Host "The value of x is:" $x
```

This will end up printing out, “The value of x is: 2.”

Write-Output

`Write-Output` prints the value of a variable to a screen, much like simply typing the variable does. The main reason to use this particular command is that you can tell it not to enumerate data. If you’re working with arrays, this can be helpful because the array will be passed down the pipeline as a single object rather than multiple objects.

An example from Microsoft’s web documentation for PowerShell showcases this perfectly. In both instances, you create an array with three values inside of it. If you measure the array, you get a count returned of 3, which makes sense. When you add `-NoEnumerate`, you get a count of 1. The three values are still there, but the array is simply being treated as one object rather than a collection of three, shown in Figure 1-13.

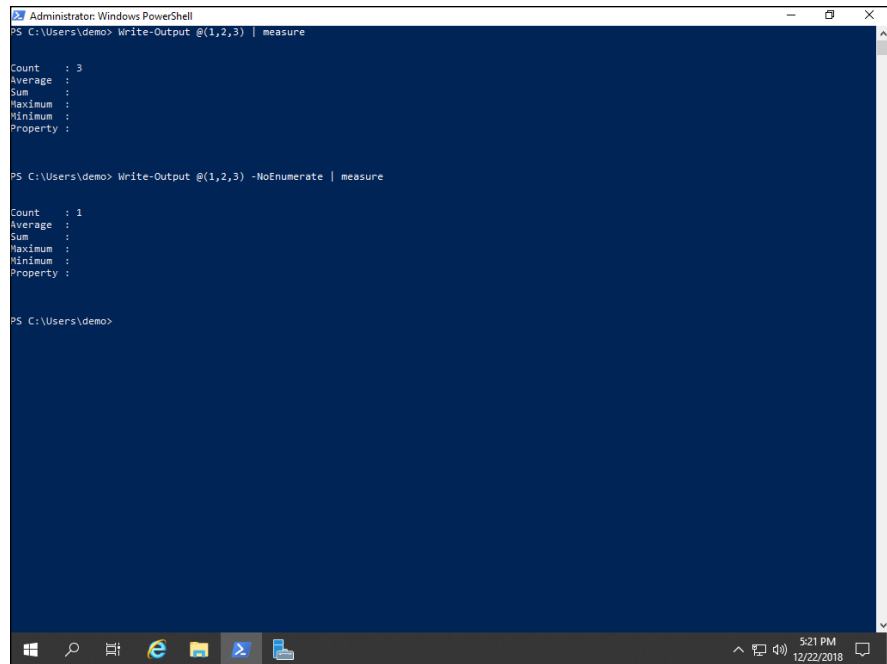
Out-File

`Out-File` is another method to export data from PowerShell into a file outside of PowerShell. This is especially helpful when you need to compile the data from a script, for instance, for analysis later on.

To get a list of processes running on a system so that you can dump them to file, you could do something like this:

```
Get-Process | Out-File -filepath C:\PSTemp\processes.txt
```

FIGURE 1-13:
Adding
-NoEnumerate
tells PowerShell
to treat arrays
differently than
normal.



Scripting logic

Loops can make a PowerShell script even more powerful. For instance, say you want to run a command against an exported list of computers from Active Directory. You can use a loop to enumerate through the imported CSV file and run the command on each individual entry.

If

The `If` statement tests a condition to see if it's true. If it is true, it will execute the code in the block. If it isn't true, it will check the next condition or execute the instructions in the final block. Here is a silly example:

```
$server = 'Windows'
If ($server -eq 'Linux') {
    Write-Host 'This is a Linux server.'
}
ElseIf ($car -eq 'Solaris') {
    Write-Host 'This is a Solaris system.'
}
ElseIf ($car -eq 'Windows') {
    Write-Host 'This is a Windows system.'
}
```

```
Else {  
    Write-Host "Don't know what kind of system this is."  
}
```

In this instance, because `$server` was initialized as Windows, the first two blocks will be skipped, but the third block will execute. The last block is ignored because it was not reached.

ForEach-Object

Using `ForEach-Object` allows you to enumerate multiple objects that have been passed through the pipeline or even imported from arrays and CSV files, just to name a few. Let's say you want to gather the names of all the processes running on your system, but you don't want any of the other information that goes along with it. You could use `ForEach-Object` to accomplish this:

```
Get-Process | ForEach-Object {$_.ProcessName}
```

While

While loops are known as pretest loops because the code is not executed if the condition set for the loop is not true. Basically, the code will be executed until the expression it's evaluating becomes false. These can be very useful when you need to increment or decrement a counter. Look at the following example:

```
$myint = 1  
DO  
{  
    "Starting loop number $myint"  
    $myint  
    $myint++  
    "Now my integer is $myint"  
} While ($myint -le 5)
```

You may be thinking, "Okay, it will count from 1 to 5." But that's not quite true. On the fifth loop, the `While` loop sees that 5 is less than or equal to 5 so it will allow it to continue. Only when it's presented with 6 will it stop execution because 6 is not less than or equal to 5. So it will count from 1 through 6. Give it a try!

Other cool tricks

There are some other neat things you can do with PowerShell, and I would be remiss in my duties as an author if I didn't add them to this chapter. Read on for more.

Exporting and importing CSV files

Exporting a CSV is very handy when you have output that contains multiple columns. This is my go-to command when I'm doing an export from Active Directory because it keeps everything neat and organized.

Here's how to export the processes on a system, similar to what you did with `Out-File` to get to a text file:

```
Get-Process | Export-Csv -Path C:\PSTemp\processes.csv
```

Importing a CSV can be the answer to a system administrator's prayers when you need to work with a lot of data and you don't want to type it in manually. Expanding on the example earlier where you exported a process list, you can import that same list and then format it nicely. Formatting is discussed in "Formatting your output" later in this chapter.

```
$Procs = Import-Csv -Path c:\PSTemp\processes.csv
```



TIP

You only need to specify `-Path` if you want to save output or import output from somewhere other than the directory you're currently in.

Exporting HTML/XML

This one isn't exactly a true export. You're actually converting the output of a command to HTML and then using `Out-File` to write it to that file. Consider the following example:

```
Get-Process | ConvertTo-Html | Out-File c:\PSTemp\processes.html
```

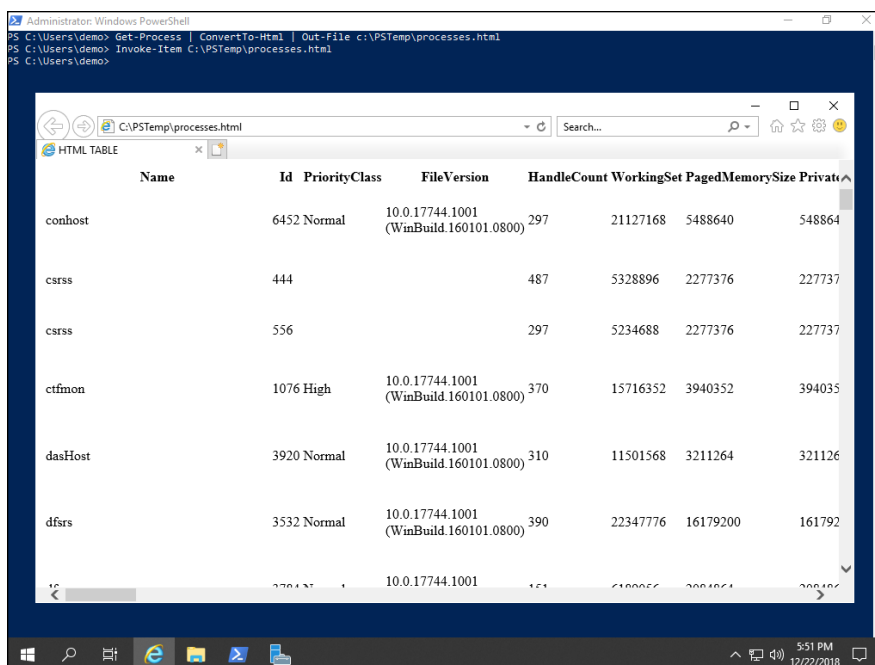
You can see this example in Figure 1-14, where I've used the `Invoke-Item` command to actually open my created HTML file that contains the output of the `Get-Process` command.

Sorting through objects

Telling PowerShell how you want it to sort objects can be very useful. For example, you may want to sort on memory usage so you can see which processes are using the most resources. The `Sort-Object` cmdlet will sort processes in the order of least CPU-intensive to most CPU-intensive:

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

FIGURE 1-14: The `ConvertTo-Html` cmdlet can be used to convert the output of a previous command to an HTML format, which can then be written to file with `Out-File`.



Filtering through objects

Examples like sorting can be very useful, but it's highly unlikely that you would want to look through all the results for CPU. You probably just want to know what the **most resource-intensive processes** are. The `Select-Object` cmdlet will return the last five results from that CPU listing, which is a much more manageable and useful list if you're troubleshooting issues.

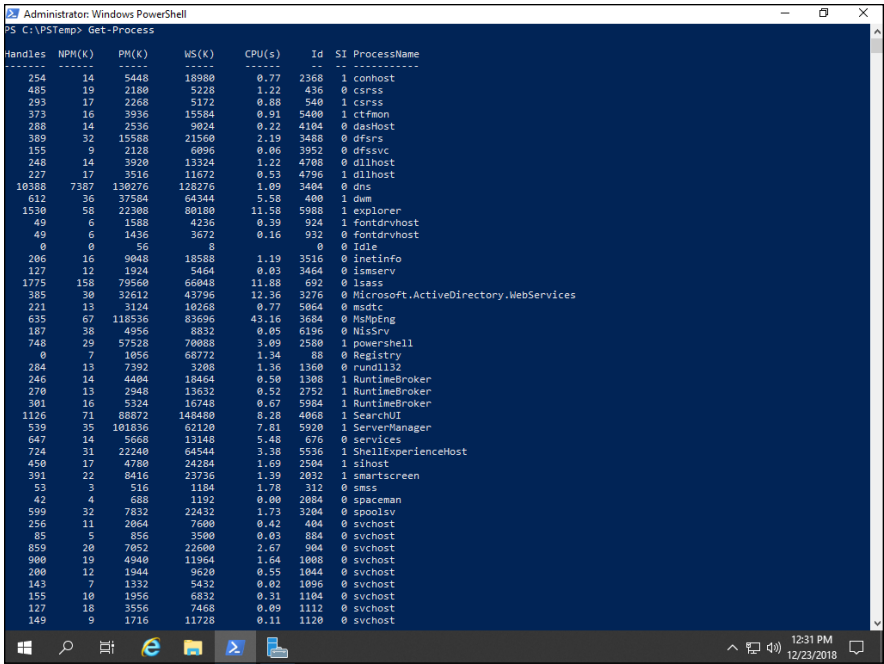
```
Get-Process | Sort-Object -Property CPU | Select-Object -Last 5
```

Formatting your output

You've probably noticed from playing in PowerShell that sometimes the output isn't as readable as it could be. So let's look at some of the ways to format output to be a little prettier.

Run the `Get-Process` cmdlet to get a baseline for appearance. Figure 1-15 gives you an idea of what the cmdlet's output looks like.

FIGURE 1-15:
The output of
`Get-Process`
is normally in a
table format.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command prompt shows the command `PS C:\PSTemp> Get-Process`. The output is a table with the following columns: Handles, NPM(K), PM(K), WS(K), CPU(s), Id, SI, and ProcessName. The table lists various system processes such as csrss, explorer, powershell, and svchost.

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
254	14	5448	18980	0.77	236	1	comhost
485	19	2180	5228	1.22	436	0	csrss
293	17	2268	5172	0.88	540	1	csrss
373	16	3936	15584	0.91	5400	1	ctfmon
288	14	2336	9824	0.22	4184	0	dashost
389	32	15588	21560	2.19	3488	0	dfsrs
155	9	2128	6096	0.06	3952	0	dfssvc
248	14	3920	13324	1.22	4708	0	dllhost
227	17	3516	11672	0.53	4796	1	dllhost
10388	7387	130276	128276	1.09	3404	0	dns
612	36	37584	64344	5.58	400	1	dwm
1530	58	22388	80180	11.58	5988	1	explorer
49	6	1580	4236	0.39	924	1	fontdrvhost
49	6	1436	3672	0.16	932	0	fontdrvhost
0	0	56	8	0	0	0	Idle
206	16	9048	18588	1.19	3516	0	inetinfo
127	12	1924	5464	0.03	3464	0	imserv
1775	158	79560	60048	11.88	692	0	lsass
385	30	32612	43796	12.36	3276	0	Microsoft.ActiveDirectory.WebServices
221	13	3124	10268	0.77	5864	0	msdtc
635	67	11836	83696	43.16	3684	0	msiexec
187	38	4956	8832	0.05	6196	0	NisSrv
748	29	57528	70088	3.09	2580	1	powershell
0	7	1856	68772	1.34	88	0	Registry
284	13	7392	3208	1.36	1360	0	rundll32
246	14	4404	18464	0.50	1388	1	RuntimeBroker
270	13	2948	13632	0.52	2752	1	RuntimeBroker
301	16	5324	16748	0.67	5964	1	RuntimeBroker
1126	71	88972	148480	0.28	4068	1	SearchUI
539	35	101836	62120	7.81	5920	1	ServerManager
647	14	5668	13148	5.48	676	0	services
724	31	22240	64544	3.38	5536	1	ShellExperienceHost
450	17	4760	24284	1.69	2504	1	slhst
391	22	8416	23736	1.39	2032	1	smartscreen
53	3	516	1184	1.78	312	0	smss
42	4	688	1192	0.00	2084	0	spaceman
599	32	7832	22432	1.73	3204	0	spoolsv
256	11	2064	7600	0.42	404	0	svchost
85	5	856	3500	0.03	884	0	svchost
859	20	7852	22600	2.67	984	0	svchost
800	19	4940	11964	1.64	1088	0	svchost
200	12	1944	9620	0.55	1044	0	svchost
143	7	1332	5432	0.02	1096	0	svchost
155	10	1956	6832	0.31	1184	0	svchost
127	18	3556	7468	0.09	1112	0	svchost
149	9	1716	11728	0.11	1120	0	svchost

FORMAT-LIST

`Format-List` takes the output of the cmdlet or whatever code is before it and formats the output into a list. The output of `Get-Process | Format-List` will look like Figure 1-16.

FORMAT-TABLE

`Format-Table` can take data in a list format and convert it into a table. With the `Get-Process` cmdlet I've been using in this chapter, it wouldn't change the output because `Get-Process` is already outputting a table. You can add the `-AutoSize` parameter so that the column sizes in the table adjust automatically.

FORMAT-WIDE

`Format-Wide` will display data in a table format, but it will only show one property of the data that it's presented. You can specify how many columns, and you can specify which property you want it to display. The example in Figure 1-17 is the output of `Get-Process` split into three columns. I haven't specified a property name, but you can see that it's using the Process Name as the property.

```

Administrator PowerShell
PS C:\PSTemp> Get-Process | Format-List

Id      : 2368
Handles: 254
CPU     : 1.4375
SI      : 1
Name    : conhost

Id      : 436
Handles: 493
CPU     : 1.21875
SI      : 0
Name    : csrss

Id      : 540
Handles: 293
CPU     : 0.90625
SI      : 1
Name    : csrss

Id      : 5400
Handles: 373
CPU     : 0.90625
SI      : 1
Name    : ctfmon

Id      : 4104
Handles: 288
CPU     : 0.21875
SI      : 0
Name    : dasHost

Id      : 3488
Handles: 385
CPU     : 2.1875
SI      : 0
Name    : dFsrS

Id      : 3952
Handles: 151
CPU     : 0.0625
SI      : 0
Name    : dFssvc

Id      : 4708
Handles: 248
CPU     : 1.21875
SI      : 0
Name    : dllhost

```

[illegible]

Running PowerShell Remotely

Running PowerShell locally can be great for automating work and simplifying administrative work. But the true strength in PowerShell comes from the fact that you can also run it remotely against other systems on your network.

Invoke-Command

`Invoke-Command` can be used locally or remotely. Because the focus of this section is on the remote usage of the cmdlet, that's where I'll focus. Say I wanted to run my favorite `Get-Process` on a system named `Server3`. The entire thing would look something like this:

```
Invoke-Command -ComputerName Server3 -Credential domain\username  
-ScriptBlock {Get-Process}
```

New-PSSession

Running `New-PSSession` allows you to establish a lasting, persistent connection to a remote system on your network. If you want to run command through your `PSSession`, you use the `Invoke-Command` cmdlet (see the preceding section). To open the new connection, type the following:

```
New-PSSession -ComputerName Server3
```

Enter-PSSession

`Enter-PSSession` allows you open an interactive session to a remote computer. The prompt changes to indicate that you're connected to the remote system. In Figure 1-18, you can see the changed prompt, as well as the commands that I ran against the remote server.

This is extremely useful if you're running headless systems like `Server Core` in your environment. To exit the interactive session, you can type **`Exit-PSSession`** or simply type **`exit`**.

```
Administrator: Windows PowerShell
PS C:\PSTemp> Enter-PSSession -ComputerName SERVER2019-DC2
[SERVER2019-DC2]: PS C:\Users\demo.SOMETESTORG\Documents> whoami
demo
[SERVER2019-DC2]: PS C:\Users\demo.SOMETESTORG\Documents> hostname
SERVER2019-DC2
[SERVER2019-DC2]: PS C:\Users\demo.SOMETESTORG\Documents> Get-Process | Format-Wide -Column 3

certsrv          csrss          csrss
dihost           dmw            fontdrvhost
fontdrvhost     Idle          inetinfo
LogonUI          lsass         msdtc
Winlogon        NlsSrv        Registry
SearchFilterHost SearchIndexer  SearchProtocolHost
services        sme           smss
SHSVchost       spoolsv       sqlwriter
svchost         svchost       svchost
svchost         svchost       svchost
svchost         svchost       svchost
svchost         svchost       svchost
svchost         svchost       svchost
VGAAuthService  vmacthlp     System
wininit         winlogon     vmtoolsd
WinPrvSE        wsmprovhost  WinPrvSE

[SERVER2019-DC2]: PS C:\Users\demo.SOMETESTORG\Documents> Exit-PSSession
PS C:\PSTemp>
```

FIGURE 1-18: Interacting with a remote server is intuitive after connecting to it with Enter-PSSession.

Getting Help in PowerShell

You can do so many things with PowerShell that it would be impossible to memorize all of them. This is where the built-in help comes in very useful. The help pages give you a description of what a command can do, along with examples and additional parameters you can use with the command.

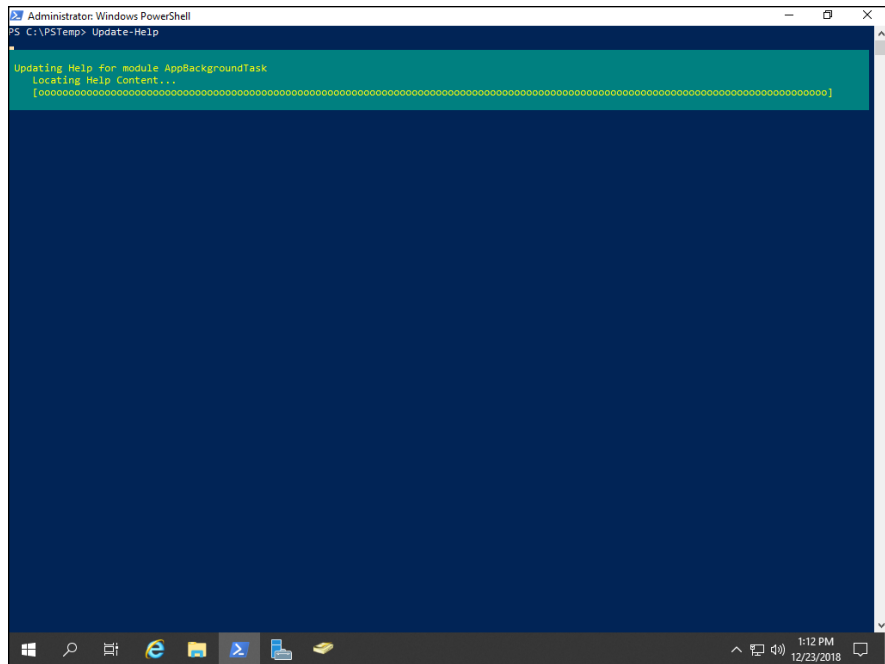
Update-Help

One of the first things I like to do with a new system is run the Update-Help cmdlet. This cmdlet pulls down the help articles available at the moment you issue the command. It can take a bit of time, so run it when you aren't needing to use it right away. In Figure 1-19, you can see what the update process looks like. Each module displays its progress. This process does require an Internet connection.



The most recent versions of the help articles are going to be on Microsoft's PowerShell Reference pages (<https://docs.microsoft.com/en-us/powershell/scripting>), not by download on your system.

FIGURE 1-19:
Updating the
help pages for
your PowerShell
modules.



Get-Help

The `Get-Help` cmdlet is like the Swiss Army knife of cmdlets. If you don't know what you're looking for, it can be very helpful. Here are some options:

- » `Get-Help Get-Process`: Running this command will return information about the `Get-Process` command. This is most useful if you need to learn more about a cmdlet, or if you need to learn more about how to interact with a cmdlet.
- » `Get-Help process`: This command is nice when you don't know the exact name of the cmdlet you're looking for. It will search for help topics that contain the word `process` and will display them to you afterward.

-Detailed and -Full

Help articles have different levels of detail that they can go into. Normally, the help article will show basic syntax to give you an idea of how to use the cmdlet. To look at more levels of detail, you can use the following:

- » `-Detailed`: The `-Detailed` parameter will display descriptions of parameters and examples of how to use them with the cmdlet.

- » -Full: The -Full parameter will truly give you all the information available regarding a cmdlet, including descriptions of parameters, examples of how to use the cmdlet, type of input/out objects, and any additional notes that are in the help file.

For example, you could type `Get-Help Get-Process -Full` to get all of the details available from the help files for the `Get-Process` cmdlet.

Identifying Security Issues with PowerShell

Given the power of PowerShell, you need to be able to secure it properly. There are several things you can do to ensure that only proper and authorized PowerShell scripts are able to run on your network.

Execution Policy

The Execution Policy allows you to define what kind of scripts are allowed to run within your network. You can set the Execution Policy through Group Policy or through the following PowerShell cmdlet:

```
Set-ExecutionPolicy -ExecutionPolicy <policy>
```

There are several policy types that can be put in place of *<policy>* in the preceding example:

- » **Restricted:** This is the default policy if no other policy is specified. It prevents PowerShell scripts from running and will not load configuration files.
- » **AllSigned:** For a script or configuration file to run, it must be signed by a trusted certificate. I cover how to do this in the next section.
- » **RemoteSigned:** This requires that any script that is downloaded from the Internet be signed by a trusted certificate. Scripts created locally do not have to be signed to run.
- » **Unrestricted:** This allows you to run all scripts and load all configuration files. You're prompted for permission before a script is run. I don't recommend using this setting.
- » **Bypass:** This is very similar to Unrestricted, except it doesn't even prompt for permission to run a script. I caution you to never use this setting.
- » **Undefined:** This removes whatever Execution Policy is currently set, unless that Execution Policy is being set through Group Policy.



WARNING

Code signing

To use an Execution Policy that is more secure, a PowerShell script needs to be signed. By signing a PowerShell script, you're validating that it came from a trusted source and that it has not been altered since it was released. If you're using `RemoteSigned`, then you only need to worry about signatures on scripts downloaded from the Internet. However, if your security people have gone wild and it's set to `AllSigned`, you need to sign your PowerShell scripts before you can run them. In the following sections, I show you the steps involved in doing that.

Creating a Code Signing Certificate

To be able to sign a PowerShell script, you need a **Code Signing Certificate (CSC)**. If you're publishing certificates for use on the Internet, you can purchase CSCs from some of the large public certificate authorities like GoDaddy and DigiCert. If you're creating certificates for internal usage, then you can create a CSC using your internal certificate authority. That's the workflow you can walk through here:

1. **Click Start and then type `certmgr.msc` and press Enter.**
2. **Right-click Personal, then select All Tasks, then Request New Certificate.**
3. **On the Before You Begin screen, click Next.**
4. **On the Select Certificate Enrollment Policy screen, choose Active Directory Enrollment Policy, and then click Next.**
5. **Select the CSC template, and then click Enroll.**

On the Certificate Installation Results screen, you should see Succeeded.

6. **Click Finish.**

Importing the certificate into the Trusted Publishers Certificate Store

If you're only going to run scripts on your local system, you can get away with manually exporting the certificate from your Personal Store and adding it to your Trusted Publishers Certificate Store. In an Enterprise situation, you'll want to use Group Policy to push your certificate to the Trusted Publishers Certificate Store on any system you'll run the script from. Follow these steps to do add your code signing certificate to your Trusted Publishers Certificate Store:

1. **Click Start, type `CertMgr.msc`, and press Enter.**
2. **Select Personal, right-click your CSC, and choose All Tasks⇨Export.**
3. **On the Welcome to the Certificate Export Wizard, click Next.**

4. On the Export Private Key screen, leave the selection on No, and click Next.
5. Leave the file format on the default .DER and click Next.
6. On the File Export screen, click Browse.
7. Select a location, name your cert, and click Next.
I'll name mine Demo CSC and save it to my Desktop.
8. On the Completing the Certificate Export Wizard, click Finish.
You get a pop-up that says "The export was successful."
9. Click OK.
10. Navigate to where you saved the certificate, and double-click it.
11. Click the Install Certificate button.
12. On the Welcome to the Certificate Import Wizard screen, select Local Machine and click Next.
13. On the Certificate Store screen, select Place All Certificates in the Following Store, and click Browse.
14. Choose Trusted Publishers and click OK; then click Next.
15. On the Completing the Certificate Import Wizard screen, click Finish.
You'll get a pop-up that says "The import was successful."
16. Click OK.

Signing your script

After you've created the certificate, and you've imported it into the Trusted Publishers Certificate Store, you can sign your certificate, and your system will trust it. So let's sign the simple Do While script that I created earlier.

```
Set-AuthenticodeSignature c:\DoWhile.ps1 @(Get-ChildItem cert:\CurrentUser\My -codesign)[0]
```

This signs the script with the certificate that is in my Personal Store. After the script is signed, I can run it. I'm not prompted and it runs without issue. If you look at Figure 1-20, you can see what the script looks like with the signature added to it.

In the example in Figure 1-21, I show you the whole thing from start to finish. I set the Execution Policy to AllSigned. Then I try to run an unsigned script. You can see I get an ugly error message saying that the script is not digitally signed. I run the PowerShell cmdlet to sign my script, and then I run the script again and it executes successfully. Pretty cool, right?

Firewall requirements for PowerShell remoting

PowerShell remoting relies on the Windows Remote Management (WinRM) service. WinRM creates two listeners, one for HTTP and one for HTTPS. To allow remote PowerShell commands to work, you need to have ports 5985 and 5986 open. Port 5985 provides HTTP support, and Port 5986 provides HTTPS support.

On an individual system, running the `Enable-PSRemoting` cmdlet does all the work needed to allow for remote PowerShell to work, including enabling the local firewall rules necessary. This cmdlet needs to be issued from an elevated (use Run as Administrator) PowerShell window.

IN THIS CHAPTER

- » Getting acquainted with the .NET Framework versions
- » Paying attention to what's new in .NET 4.7
- » Viewing the contents of the Global Assembly Cache
- » Examining .NET Standard and .NET Core

Chapter 2

Understanding the .NET Framework

Many people wonder why they need to understand the .NET Framework and, more important, why it's bundled with PowerShell. Using .NET, not only can you do things in PowerShell that you wouldn't be able to do natively, but you also can create functions and modules with the .NET code that you can reuse.

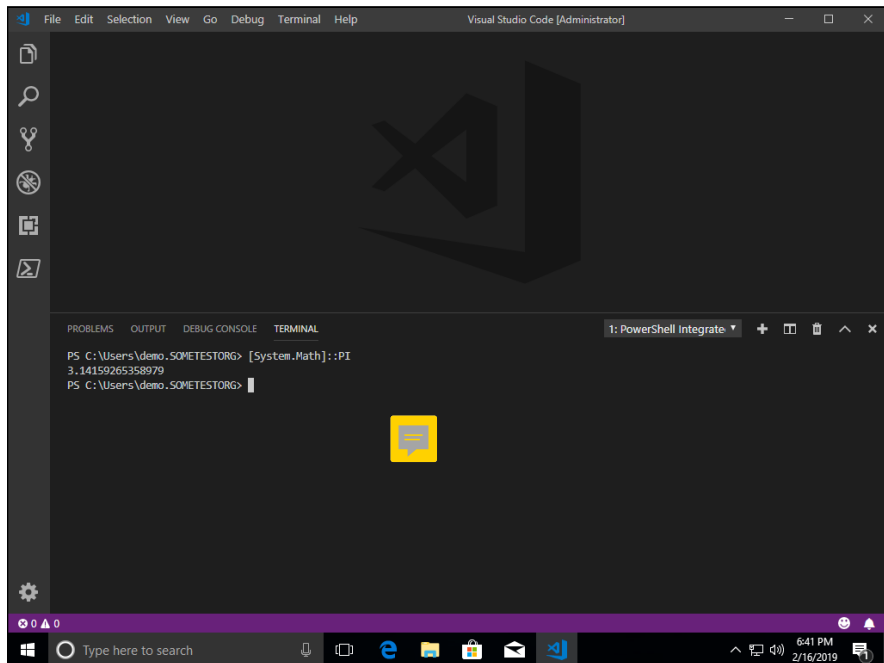
In this chapter, I explain the basics of .NET and how to interact with it. I don't cover .NET programming — that's a whole other book's worth of material.

Introducing the Various Versions of .NET Framework

Before I jump into .NET versions, I want to make sure that you understand what the .NET framework actually is. A framework allows a programmer to call code instead of having to write the code each time the programmer wants the functionality. The .NET framework gives developers the code they need to write

.NET applications without having to custom develop every single little piece of code themselves. .NET is integrated with PowerShell, so you can call the same snippets of .NET code that developers can from within PowerShell, either in the console or in a script. For instance, Figure 2-1 shows a piece of .NET code called from PowerShell that displays the value of pi.

FIGURE 2-1:
The .NET Framework expands the functionality of PowerShell greatly with code that can be called on the console or via script.



Each new version of the .NET Framework adds new functionality and fixes old problems. The .NET Framework follows a similar cadence to most products in that it has major and minor versions. The major releases tend to focus heavily on new features, while the minor versions add features and fix issues found in the previous releases.

As of the time of this writing, version 4.7 is the current major version, and 4.7.2 is the current minor version. The 4.7 versions shipped with Windows Server 2019; version 4.6.2 shipped with Windows Server 2016; and version 4.5.1 shipped with Windows Server 2012 R2. It's very common to have multiple versions of the .NET Framework installed on the same system; there is usually no issue with them co-existing.



TIP

The version of .NET you're currently on is stored in the Windows Registry. To locate the release number, you can use the following command:

```
(Get-ItemProperty 'HKLM:\SOFTWARE\Microsoft\NET Framework Setup\
NDP\v4\Full' -Name Release).Release
```

This command returns a number. Table 2-1 lists the minimum number for each major version. When I run this command on my system, for example, I get 461814. According to Table 2-1, that means that I have version 4.7.2, which is correct because the minimum value for 4.7.2 is 461808.

TABLE 2-1

.NET Versions with Release Values

Version	Minimum Value
.NET Framework 4.5	378389
.NET Framework 4.5.1	378675
.NET Framework 4.5.2	379893
.NET Framework 4.6	393295
.NET Framework 4.6.1	394254
.NET Framework 4.6.2	394802
.NET Framework 4.7	460798
.NET Framework 4.7.1	461308
.NET Framework 4.7.2	461808

Source: https://docs.microsoft.com/en-us/dotnet/framework/migration-guide/how-to-determine-which-versions-are-installed#ps_a



TIP

Most of the time, you have one version of .NET installed, and that's it. Sometimes, though, you may have a legacy application that needs an older version of .NET, and a newer application that requires a newer version of .NET. The great thing about .NET is that you can run more than one version side by side. For instance, your legacy application may need .NET 3.5, but your newer application needs .NET 4.5. You can install both versions of .NET on your system, and each application will be able to run with the version of .NET that it needs.

.NET is available in the server operating system as a feature. To install .NET, follow these steps:

1. In Server Manager, choose **Manage** ⇨ **Add Roles and Features**.
2. On the **Before You Begin** screen, click **Next**.
3. On the **Select Installation Type** screen, choose **Role-Based or Feature-Based Installation**, and then click **Next**.
4. On the **Select Destination Server** screen, click **Next**.
5. On the **Select Server Roles** screen, click **Next**.
6. On the **Select Features** screen, select either **.NET Framework 3.5 Features** (which installs .NET Framework 2.0 and 3.5) or **.NET Framework 4.7**, and then click **Next**.
7. On the **Confirm Installation** screen, click **Install**.

Focusing on New Features in .NET 4.7

.NET version 4.7 introduced new features in a few key areas:

- » **Core:** Among the core functionality that was added in version 4.7 is increased functionality for elliptic curve cryptography (ECC) and some other improvements for cryptography in general.

Core also includes better support for control characters and is enabled by default for any application that targets .NET 4.7. It can be opted in to with applications that are using an older version of .NET.
- » **Networking:** Developers no longer need to hard-code a Transport Layer Security (TLS) version into their applications. Instead, they can target the TLS protocols that are supported by default on the installed operating system.
- » **ASP.NET:** ASP.NET is used to build web applications and offers three different frameworks to support that objective — Web Forms, MVC (model, views, and controllers), and Web Pages. ASP.NET got a newer feature with .NET 4.7 called Object Cache Extensibility, which allows developers to make changes to the in-memory object caching and monitoring if the default implementation does not meet their needs.
- » **Windows Communication Foundation (WCF):** WCF allows you to send messages between your services and your clients. With .NET 4.7, WCF got the ability to configure message security settings to use the stronger TLS 1.1 and 1.2 rather than SSL 3.0 and TLS 1.0. It's an opt-in setting so you have to go in

and enable it inside of your application's configuration file. Plus, in .NET 4.7, several changes were made to improve the reliability and stability in relation to serialization options.

- » **Windows Forms:** Windows Forms allows you to build graphical ways for your application to interact with your users. It allows you to add controls to your forms, and to tell the application what to do when a user clicks a button. As of .NET 4.7, Windows Forms can now support high-DPI monitors. This is an opt-in feature that must be enabled within your application.
- » **Windows Presentation Foundation (WPF):** WPF is a UI-based framework that aids in the creation of desktop applications. WPF now supports the touch/stylus stack when using WM_POINTER messages, instead of Windows Ink Services Platform. This is an opt-in feature, which you have to enable inside of your application. Printing APIs were also updated.

Viewing the Global Assembly Cache

Before I dive into the Global Assembly Cache (GAC), you may be wondering what it is. The GAC is responsible for storing assemblies that are shared by multiple applications on a computer. The *assembly*, at its most basic definition, is an executable of some kind. It contains all the code that will be run and serves as the boundary for the application. Many assemblies are installed when .NET is installed on your system.

.NET Framework versions 4 and up store their assemblies in %windir%\Microsoft.NET\assembly. The %windir% is a placeholder for the Windows directory, which is typically located at C:\Windows.

Viewing the assemblies in the GAC is done using a tool called `gacutil.exe`. This tool is a part of the Developer Command Prompt for Visual Studio so you need to install Visual Studio on your system if you want to play with the GAC on your system. Visual Studio IDE Community is the free version and does include the Developer Command Prompt. Figure 2-2 gives you an idea of what the GAC looks like when you view the assemblies. I've typed the following command:

```
gacutil.exe -l
```

This command lists all the assemblies within the GAC.

FIGURE 2-2:
You can view
the contents of
the GAC with
the Developer
Command
Prompt.

```

Administrator: Developer Command Prompt for VS 2017
** Visual Studio 2017 Developer Command Prompt v15.9.4
** Copyright (c) 2017 Microsoft Corporation
*****
C:\Program Files (x86)\Microsoft Visual Studio\2017\Community>gacutil -l
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

The Global Assembly Cache contains the following assemblies:
Microsoft.Ink, Version=6.1.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=AMD64
Microsoft.Interop.Security.AzRoles, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=AMD64
mswacdm, Version=1.8.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=AMD64
smmlib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=AMD64
Microsoft.Ink, Version=6.1.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=x86
Microsoft.Interop.Security.AzRoles, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=x86
mswacdm, Version=1.8.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=x86
smmlib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=x86
Microsoft.Ink.Resources, Version=6.1.0.0, Culture=en, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.ManagementConsole, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.ManagementConsole.Resources, Version=3.0.0.0, Culture=en, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.Commands.Diagnostics, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.Commands.Diagnostics.Resources, Version=1.0.0.0, Culture=en, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.Commands.Management, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.Commands.Management.Resources, Version=1.0.0.0, Culture=en, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.Commands.Utility, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.Commands.Utility.Resources, Version=1.0.0.0, Culture=en, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.ConsoleHost, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.ConsoleHost.Resources, Version=1.0.0.0, Culture=en, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.Security, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL
Microsoft.PowerShell.Security.Resources, Version=1.0.0.0, Culture=en, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL

```

Understanding assembly security

Because the GAC lives in the Windows folder, it inherits the permissions of the Windows folder. In many cases, you may want to tighten the permissions on the GAC directories so that only administrators can delete assemblies. If someone deletes an assembly that the system or an application relies on to function properly, that application will no longer work.

Identifying the two types of assembly privacy

Two types of assemblies make up the .NET presence on your system:

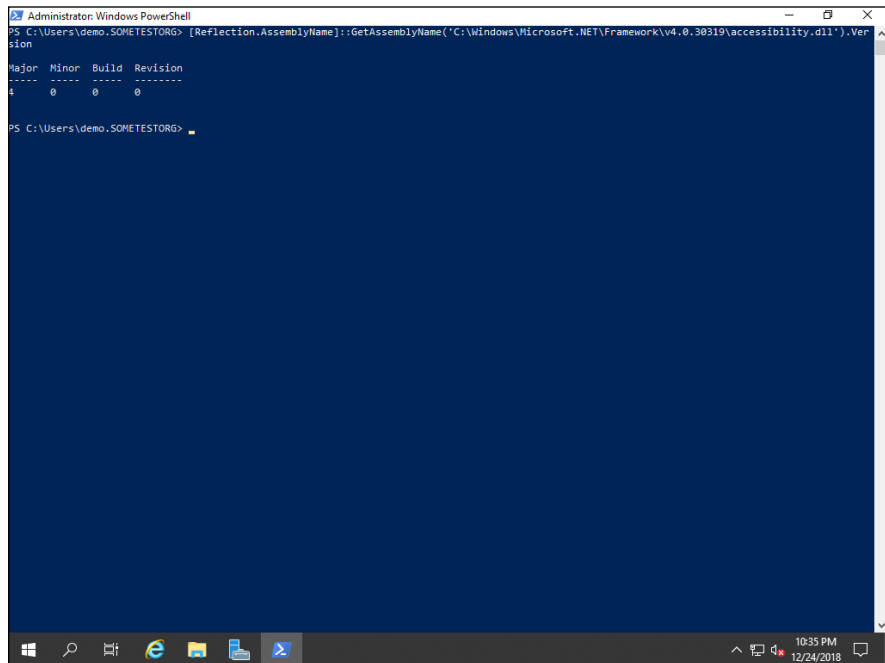
- » **Private:** Private assemblies are deployed with an application and can only be used by that application. Think of them like the children on the playground who won't share.
- » **Shared:** Shared assemblies are available to be used by multiple applications on your system. They're stored in the WinSxS folder and are installed via Windows Update and Windows Installer packages.

Viewing assembly properties

In older versions of the Windows Server operating system, you could simply right-click an assembly to get all the properties of the file. That function was removed several operating system versions ago. Now if you need to get information on the assembly file, your best bet will be to go through PowerShell. Say that I want to view the version information on the `accessibility.dll` that is in use. This is the command that I would need to run:

```
[Reflection.AssemblyName]::GetAssemblyName('C:\Windows\Microsoft.NET\Framework\v4.0.30319\accessibility.dll').Version
```

After I've run this command I'm presented with the major, minor, build, and revision numbers, as shown in Figure 2-3.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is `[Reflection.AssemblyName]::GetAssemblyName('C:\Windows\Microsoft.NET\Framework\v4.0.30319\accessibility.dll').Version`. The output is a table with four columns: Major, Minor, Build, and Revision. The values are 4, 0, 30319, and 0 respectively.

Major	Minor	Build	Revision
4	0	30319	0

FIGURE 2-3:
You need to use PowerShell to view the assembly properties in Windows Server 2019.

Understanding .NET Standard and .NET Core

The .NET Framework has been a staple for many years, but newer frameworks are gaining in popularity.

.NET Core

.NET Core is one of the newest members of the .NET Framework family. It's open source and it can be run on Windows, Linux, and macOS. With .NET Core, you can build applications that are cross-platform. If your application is developed with .NET Core, then only .NET Core applications will be compatible — you won't be able to support Xamarin or the classic .NET Framework. .NET Core is an implementation of the specifications that are set in the .NET Standard.

You may wonder why you would use .NET Core if it isn't compatible with the other runtimes. There are a couple good reasons:

- » You can develop .NET Core on Windows, Linux, or macOS.
- » If you're coding for a mobile application, .NET Core is optimized for mobile work.

.NET Standard

.NET Standard is a set of APIs that all the .NET frameworks must support. This includes .NET Core, Xamarin, and the classic .NET Framework. It's important to note that .NET Standard is a specification, not a framework. It's used to build libraries that can be used across all your .NET implementations, including the traditional .NET Framework, the newer .NET Core, and Xamarin.

Tying it all together: .NET and PowerShell

PowerShell Core 6.0 uses the newer .NET Core as its runtime. This means that you can now run PowerShell on Windows, Linux, and macOS. PowerShell Core also enables you to take advantage of all the awesome .NET Core APIs in your commands and scripts, which really extends the utility and capabilities of your scripts. You can start working with PowerShell Core without impacting your current installation of PowerShell because both PowerShell and PowerShell Core can be run side-by-side. PowerShell Core is available for download from the PowerShell repo on GitHub at <https://github.com/PowerShell/PowerShell/releases>.

- » Examining common scripts and cmdlets
- » Running PowerShell scripts or cmdlets
- » Working remotely with PowerShell
- » Performing administrative tasks with PowerShell scripts

Chapter 3

Working with Scripts and Cmdlets

In Chapter 1, I explain what PowerShell is and fill you in on the basics of working with it. In Chapter 2, I tell you a little bit about .NET and its interaction with PowerShell. This chapter is all about putting that information and your skills to use by starting to build out the scripts that you'll use on a daily basis as a system administrator.

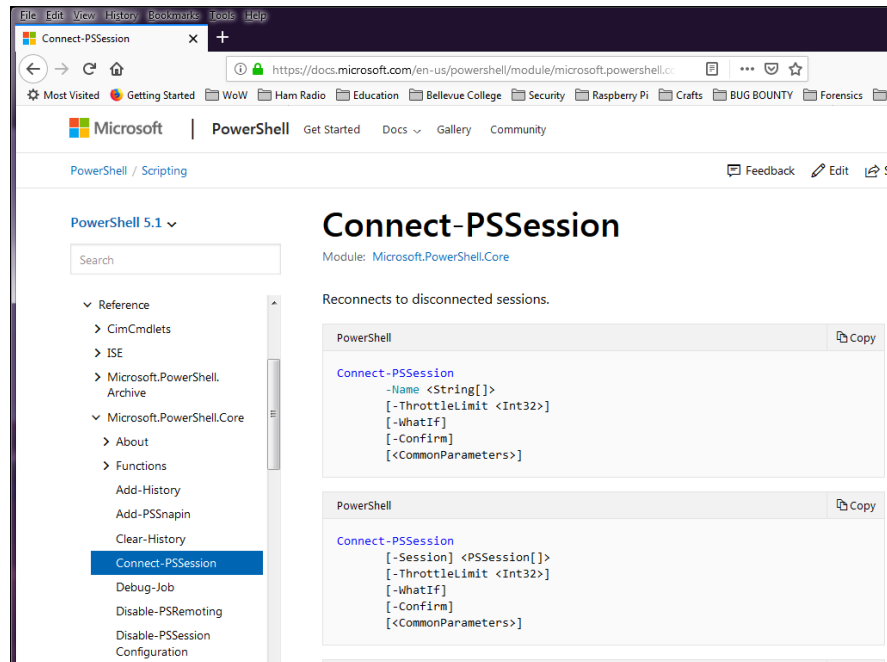
Introducing Common Scripts and Cmdlets

Writing single lines of PowerShell is pretty common when you have administrative tasks that you're trying to perform, but most of the really useful and powerful PowerShell comes from the ability to put multiple cmdlets into a script.

The Microsoft website has some great reference material for PowerShell. What I find the most beneficial is the Reference link off the main PowerShell site (<https://docs.microsoft.com/en-us/powershell/scripting/>). When you click the Reference link, you get all the Help pages that you can request from the PowerShell console but in a nice graphical interface. You can find syntax help and examples of how to use the cmdlets.

Each cmdlet is grouped by type, and then listed for you to look through or select. Figure 3-1 shows you an example of what the Connect-PSSession page looks like.

FIGURE 3-1:
The Reference
page from
Microsoft's
website contains
hundreds of
cmdlets with
descriptions,
syntax, and
examples.



If you want to get more examples to work with, the PowerShell Gallery has thousands of example scripts that you can download and start using. What I love about PowerShell Gallery is that you can search for what you want to download. For instance, I've downloaded PowerShell modules written for enterprise-grade firewalls. The vendors only published an API; the community came together and wrote a PowerShell module that allows me to use familiar PowerShell syntax rather than learn the vendor's API.

Executing Scripts or Cmdlets

For the most part, executing a PowerShell script is not that much different from using a Command Prompt. You can right-click it and choose Run with PowerShell, or you can call it from the PowerShell console. Don't forget about the execution policy that was discussed in earlier chapters. If your execution policy is not set to

where it will allow the desired script to run, then PowerShell will give you an error when you try to execute the script.

When calling it from the console, you can give it the location to the script:

```
C:\PSTemp\dowhile.ps1
```

Or you can use the `.` to indicate that you're already in the correct directory:

```
.\dowhile.ps1
```

Working with COM objects

You'll more commonly work with the .NET framework when writing scripts in PowerShell, but you can also work with COM objects. The commands are very similar, but there are syntax differences in how you identify COM objects. This example creates a COM object that represents Internet Explorer, and sets the visible property to `$True` so that you can see the Internet Explorer window. It sets the URL to the `www.dummies.com` website.

```
$IE = New-Object -COMObject InternetExplorer.Application  
-Property @{Navigate2="www.dummies.com"; Visible = $True}
```

If you're wondering what properties you have to work with, you can find them as you would with a regular cmdlet. The previous example, for instance, could be typed as follows, to find out the properties that are available:

```
$IE = New-Object -COMObject InternetExplorer.Application  
$IE | Get-Member
```

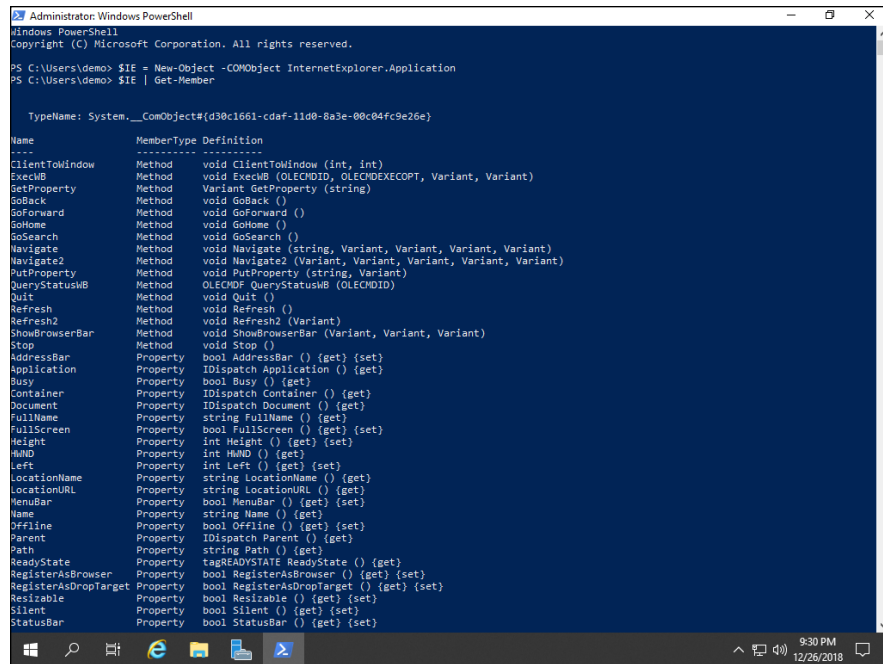
Figure 3-2 shows you an example of what the display from `Get-Member` would be. You can see `Navigate2` shows up as one of the methods, and if you scrolled down you would find `Visible` toward the bottom of the list of properties.

Combining multiple cmdlets

Using one cmdlet at a time is not an efficient use of your time, and let's face it, every system administrator wants to be able to save some time.

You've learned about the pipeline and about variables. All these things will enable you to be a PowerShell ninja, one idea at a time.

FIGURE 3-2: Checking the properties of COM objects is similar to checking the properties of cmdlets, using `Get-Member`.



For instance, you could write this line of code in this manner:

```
$cmd = Get-Command
$cmd | Get-Member
```

Or you could simply combine the two from the beginning like this:

```
Get-Command | Get-Member
```

Combining multiple cmdlets instead of running them one at a time, and storing their values into variables, works much better and far more efficiently. In Chapter 4 of this minibook, I show you how to take this to the next level and save your commands into a file that you can write anytime. PowerShell scripts are absolutely wonderful. They're huge time savers — and, if designed correctly, you can use them for years to automate processes and batch jobs.

Working from Another Location

You may need to work with data remotely from time to time without the requirement of logging in to the remote system to do that. There are several methods to accomplish this task. You may also want to be able to get information through the remote console.

To list the items in a share, similar to running the `dir` or `ls` commands:

```
Get-ChildItem \\servername\sharename
```

To check the health of your file shares:

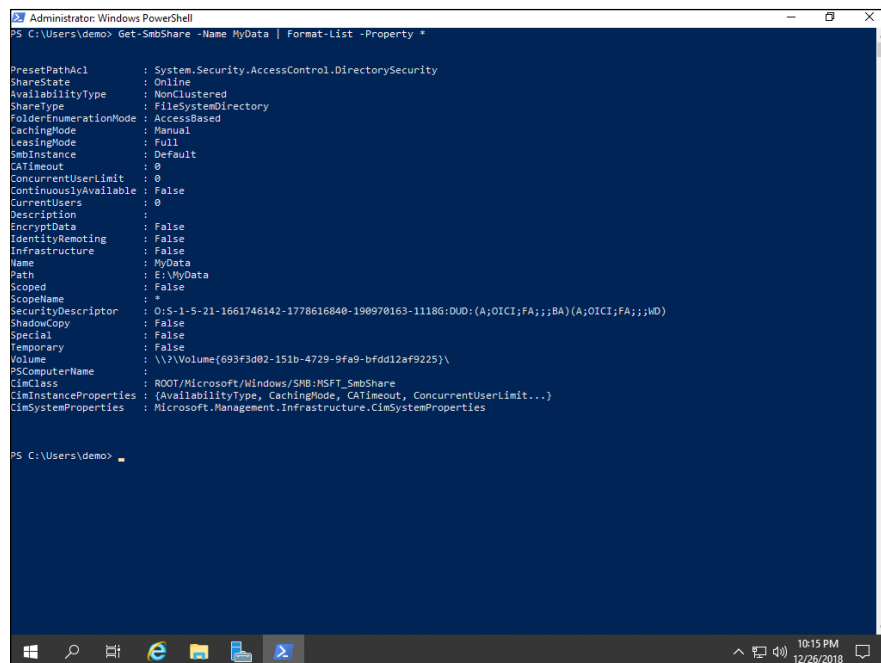
```
Get-FileShare -FileServer (Get-StorageFileServer -FriendlyName  
"servername")
```

Working with Server Message Block (SMB), which is a file-sharing protocol for use over a network, is simple as well. Let's look at a few commands that will allow you to work with SMB shares on Windows file servers. These run locally so you will need to create a session with the system you want to run this against.

`Get-SmbShare` returns a list of all the SMB shares on the local system. This can be helpful when you need to go a step further and list the properties of a share. For example, say I want to see the properties of a share named `MyData`. I can type something like the following:

```
Get-SmbShare -Name "MyData" | Format-List -Property *
```

As you can see in Figure 3-3, you can get some valuable information on your share with that simple command. You can find out the physical path, whether the data is encrypted, whether shadow copies are turned on, and so on.



```
Administrator: Windows PowerShell
PS C:\Users\demo> Get-SmbShare -Name MyData | Format-List -Property *

PresetPathAcl      : System.Security.AccessControl.DirectorySecurity
ShareState         : Online
AvailabilityType   : NonClustered
ShareType          : FileSystemDirectory
FolderEnumerationMode : AccessBased
CachingMode        : Manual
LeasingMode        : Full
SmbInstance        : Default
CTimeout           : 0
ConcurrentUserLimit : 0
ContinuouslyAvailable : False
CurrentUsers       : 0
Description        :
EncryptData        : False
IdentityRemoting   : False
Infrastructure     : False
Name               : MyData
Path               : E:\MyData
Scoped             : False
ScopeName          : *
SecurityDescriptor : O:S-1-5-21-1661746142-1778616840-190978163-11186:DUD:(A;OICI;FA;;;BA)(A;OICI;FA;;;WD)
ShadowCopy         : False
Special            : False
Temporary          : False
Volume             : \\.\Volume{693F3D02-151b-4729-9fa9-bfdd12af9225}\
PSComputerName     :
CimClass           : ROOT\Microsoft\Windows\SMB\NSFT_SmbShare
CimInstanceProperties : (AvailabilityType, CachingMode, CTimeout, ConcurrentUserLimit...)
CimSystemProperties : Microsoft.Management.Infrastructure.CimSystemProperties

PS C:\Users\demo>
```

FIGURE 3-3:
`Get-SmbShare`
is a very powerful
cmdlet that
gives you the
ability to collect
a great deal of
information.

Performing Simple Administrative Tasks with PowerShell Scripts

In this section, I show you some cool things that you can do with PowerShell scripts that might help you right now.



REMEMBER

Keep in mind that for some of these scripts you may need to install a module to get it to work properly. If you run the Active Directory scripts on your domain controller, you won't need to add the AD module. If you're running from a system without Active Directory (like your desktop, for example), you need to import the Active Directory module with the command `Import-Module ActiveDirectory`. If you have the Remote Server Administration Tools (RSAT) installed on your desktop, the modules will be able to automatically load when needed.

Adding users in Active Directory

Adding users in Active Directory is a pretty common task. You can do this from the graphical user interface (GUI), but this method can be much faster. This example creates a user named George Smith, adds him to the Sales OU, prompts for the password (which is stored securely), enables the account, and ensures that the user will need to change his password after he has logged in.

```
New-ADUser -Name " George Smith" -GivenName "George" -Surname  
"Smith" -SamAccountName "gsmith" -UserPrincipalName "gsmith@  
sometestorg.com" -Path "OU=Sales,DC=sometestorg,DC=com"  
-AccountPassword(Read-Host -AsSecureString "Input Password")  
-Enabled $true -ChangePasswordAtLogon $true
```

As you can see in Figure 3-4, the user account is created and placed into the OU that I specified. The account is enabled and will be ready for the user on his first day.

Creating a CSV file and populating it with data from Active Directory

Pulling data from Active Directory is an important skill for a system administrator to have. The following code queries Active Directory for Server operating systems. You can further define your filter to return specific versions of the server OS as well. Notice in Figure 3-5 that the information from onscreen appears in the CSV. There are no service packs out yet, so the `OperatingSystemServicePack` field is empty.

FIGURE 3-4:
Creating a user
in PowerShell
is quick, and
the user shows
up almost
instantaneously,
as shown in
the Active
Directory Users
and Computers
window.

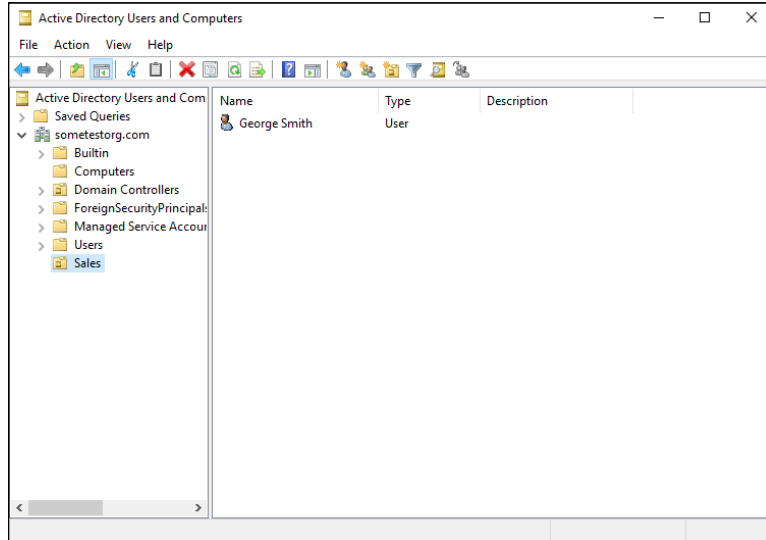
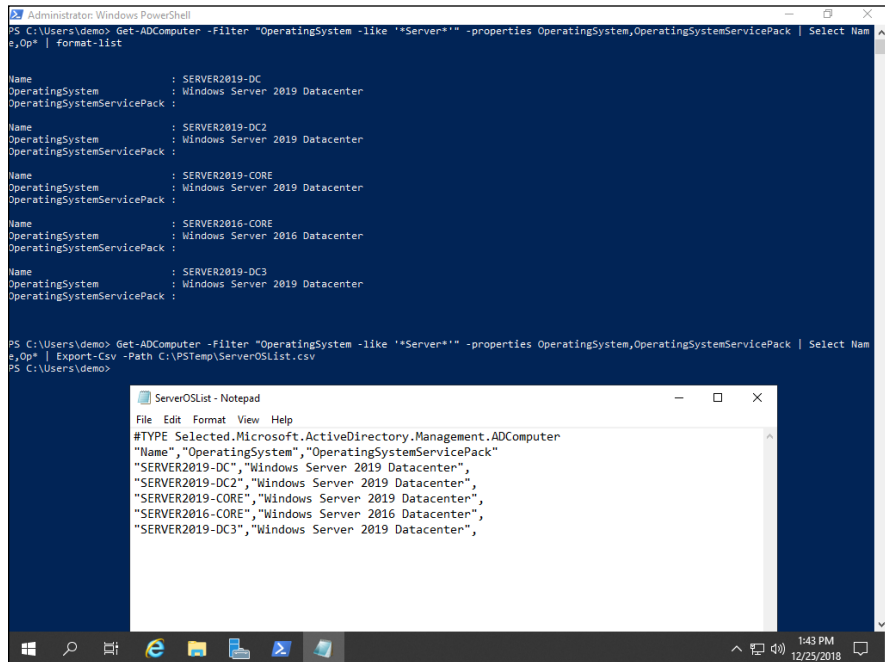


FIGURE 3-5:
Running your
query and having
the information
on the screen
can be nice, but
it's even better
when it's in a file
you can work
with later.



```

Get-ADComputer -Filter "OperatingSystem -like '*Server*'"
-Properties OperatingSystem,OperatingSystemServicePack |
Select Name,Op* | Export-CSV c:\PSTemp\ServerOSList.csv
  
```

Checking to see if a patch is installed

It never fails — when a big vulnerability is released and a patch is announced, your boss is almost always going to ask, “Is the patch installed?” If you work for a larger organization, you may have purchased a tool that can manage this for you. If you work for a smaller org, though, or if you want to spot-check the accuracy of your tools, PowerShell does give you a simple way to check to see if a patch is installed. You can use this with `Invoke-Command`, which is great if you want to check it against a list of all your systems, for example.

```
Get-Hotfix -Id "KBXXXXXX" -ComputerName <servername>
```

Replace the Xs with the numbers in the actual Knowledge Base (KB), and the *servername* with the remote server’s hostname. If you’re running this command locally, you can omit the `-ComputerName` parameter altogether.

Checking running processes or services

Last but certainly not least are the administrative cmdlets to check for running processes and services.

`Get-Process` will return all the active processes by default. You can run it by itself, and you receive a table-formatted output with the active processes.

`Get-Service` is a little different in that it’s totally normal to have services that are not running during the day. If you want to see only the running processes, you need to type a command similar to the following:

```
Get-Service | Where-Object {$_.Status -eq "Running"}
```

The `$.Status` is a property of `Get-Service`. Using `Where-Object` in front of it will filter your output so that you only see running services.

What do you do if you want to check the services on multiple systems? You select multiple system names like this:

```
Get-Service -Name "WinRM" -ComputerName Server2019-DC,  
"Server2019-dc2", "Server2019-dc3" | Format-Table -Property  
MachineName, Status, Name, DisplayName -auto
```

This command will check for WinRM specifically on the three servers that I have included after the `-ComputerName` parameter.

- » Understanding and Creating PowerShell scripts
- » Setting an execution policy for your PowerShell script
- » Signing a PowerShell script with a Code Signing Certificate
- » Creating a PowerShell advanced function with VS Code

Chapter 4

Creating Your Own Scripts and Advanced Functions

Tons of scripts are already purpose made and ready for download, but nothing is quite as satisfying as writing your own scripts and cmdlets. That feeling of accomplishment really can't be beat when the script you've been working on is put to use.

With Infrastructure as Code (IaC) gaining in popularity, being able to write custom scripts helps to make you a more marketable employee, not just from a system administration standpoint, but also from a DevOps standpoint. IaC allows you to script the deployment of a server so that you can respond to the need to scale much more quickly than if you had to manually build a server.

In this chapter, I show you how to create your own custom components of PowerShell. You discover how to create new shell extensions and your own PowerShell scripts, and you find out about creating your own cmdlets.

Creating a PowerShell Script

You've got this monotonous task you have to do every single day. This task regularly take you an hour to complete. You want to reclaim your hour. What do you do? You build a PowerShell script, of course!

PowerShell is probably one of my favorite scripting languages. The main reason is that when you have a grasp of the syntax and how to look up the properties of various cmdlets, you're really only limited by your imagination.

Before I move on to working with scripts and functions, let's get a few definitions out of the way:

- » **Script:** A script is a series of commands that are run in order, dependent on whether you have conditional operations occurring within your script. Scripts often contain cmdlets, loops, and other elements that, when run together, accomplish some task. Scripts are the easiest method to automate repetitive work.
- » **Advanced function:** An advanced function allows you to create and do the same things you can do with cmdlets, but without having to learn .NET Framework languages like C#, and without having to compile your code. You do have to follow the naming rules for PowerShell commands when creating a function — it should consist of the same verb/noun syntax as the usual PowerShell cmdlets.

Creating a simple script

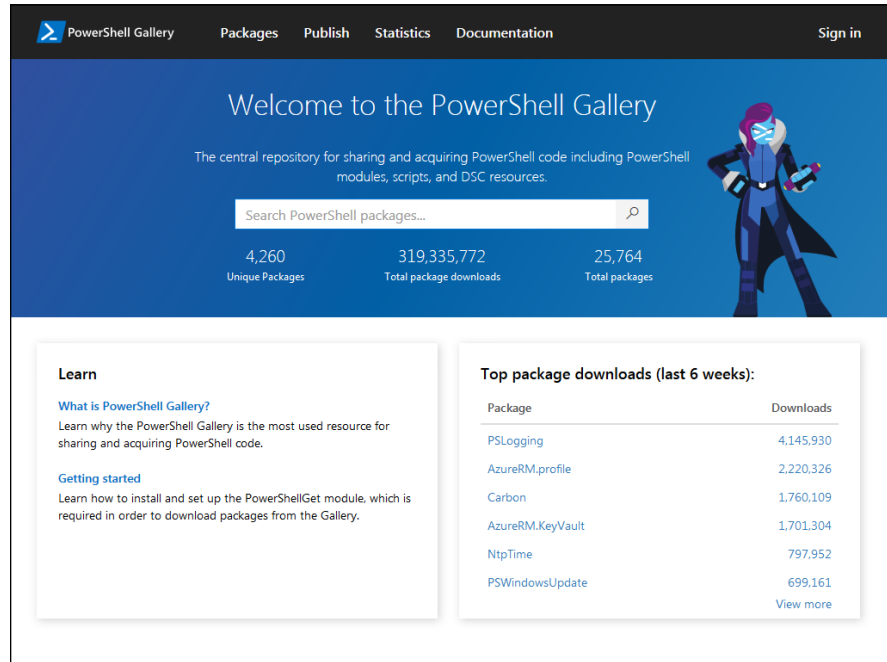
When creating your first script, start with a small goal in mind and then build on it from there. If you try to do something really large and detailed, you may run into issues and get frustrated. Look at examples on the Internet to see how others have solved for the same issues, and experiment.

PowerShell scripts are always saved with a filetype of PS1. This tells your system that the file is a PowerShell script file and will suggest PowerShell or your favorite text editor to edit it or will open PowerShell to execute it.

If you ever find yourself in need of samples or you want to download pieces of code so that you're crafting code from scratch, I highly recommend PowerShell Gallery (www.powershellgallery.com). It's maintained by Microsoft and has modules, scripts, and even some examples of PowerShell DSC, which I talk about in Chapter 5 of this minibook. Figure 4-1 shows you the PowerShell Gallery homepage. At the time of this writing, you can see that there are 4,260 unique

scripts and modules available from PowerShell Gallery. What I really like is that you can see how many downloads of each resource have been made, and any questions or comments that have been made to the authors of the scripts or modules.

FIGURE 4-1: PowerShell Gallery is an excellent resource to download or see examples of various scripts or modules.



Before I get into an example of a script, I want to add that you should always use comments when you're writing a script. You can denote a comment in PowerShell by starting the line with a hash (#). Everything after the hash on that line is ignored and is treated like a comment. Comments are useful for you because they can form an outline of what you're trying to accomplish. They can serve as documentation for you later and can assist your co-workers in either using the script or helping you write it.

In Chapter 3 of this minibook, I show you an example of how to create a new user in Active Directory using PowerShell instead of using the graphical Active Directory Users and Computers. The real power of that little bit of code becomes apparent when it's put into a script that can take a CSV file (from HR perhaps) and import it into PowerShell, and then have PowerShell loop through each row and create each user.

In the following sections, I step through this script step by step. Then I show you the whole thing all together. Never fear! The script is on the GitHub repo created

for this book, and you can download it from there, rather than having to retype it. The GitHub repository created for this book is located at <https://github.com/sara-perrott/Server2019PowerShell>.

Creating the CSV file

The CSV file is the most important piece of this whole exercise because it provides the input to the script. The column names are assigned in the first part of the script to variables so that they can be called when you get to the loop that processes each row in the CSV. Figure 4-2 shows you a sample of what the CSV should look like. After the CSV is complete, you can move on to the script.

	A	B	C	D	E	F	G	H	I	J	K
1	Fname	Lname	Username	Email	Phone	Dept	Password	Title	OU		
2	Moe	Stoooge	Mstoooge	mstoooge@sometestorg.com	555-555-1234	Sales	Changem3!	Sr. Salesman	OU=Sales,DC=sometestorg,DC=COM		
3	Larry	Stoooge	Lstoooge	lstoooge@sometestorg.com	555-555-1235	Sales	Changem3!	Salesman	OU=Sales,DC=sometestorg,DC=COM		
4	Curly	Stoooge	Cstoooge	cstoooge@sometestorg.com	555-555-1236	Sales	Changem3!	Salesman	OU=Sales,DC=sometestorg,DC=COM		
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											
15											
16											
17											
18											
19											
20											
21											
22											
23											
24											
25											
26											
27											

FIGURE 4-2:
Using CSV files
to import data
sets for scripts is
a simple way to
deal with multiple
inputs.



TIP

As a working security professional, I have to point out that storing passwords in plaintext CSV files is not a good thing to do. I suggest setting complex temporary passwords and destroying the CSV when it's complete. You can, of course, send the credentials to the supervisor of the new employee by whatever means is approved in your organization before destroying it. Don't leave spreadsheets hanging around with passwords in them, though. They're a treasure trove for hackers.

Creating the script

This script imports users from the CSV file you created earlier and creates an Active Directory account for each of them.

Assuming you aren't working on a domain controller, you need to import the Active Directory module so that you can work with the AD cmdlets. This process used to be manual, but now it's automatically imported when it's needed. Assuming that you have Remote Server Administration Tools (RSAT) installed, the module for Active Directory will load when you use a cmdlet that requires it.

The following line of PowerShell will import the CSV file when run and will store the contents in the `$ImportADUsers` variable.

```
$ImportADUsers = Import-Csv C:\PSTemp\UserImport.csv
```

The next line starts a `foreach` loop. This loop will go row by row through the CSV file. Each row is stored in the `$User` variable. Each of the column names is called by the `$User.columnname` section, and the value is stored in each of the variables that matches the same name. (Take note of the curly brace in between the `foreach` and the variable block.)

```
foreach ($User in $ImportADUsers)
{
    $FName = $User.FName
    $LName = $User.LName
    $Username = $User.username
    $Email = $User.Email
    $Phone = $User.Phone
    $Dept = $User.Dept
    $Password = $User.password
    $Title = $User.Title
    $OU = $User.OU
}
```

Next you want to check if the username you're creating already exists in Active Directory. The simplest way to accomplish this is with an `If...Else` statement. So, here's the `If` portion, which checks if the user account already exists, and if it does, prints the error to the screen:

```
if (Get-ADUser -Filter {SamAccountName -eq $Username})
{
    Write-Warning "This user account already exists in Active
    Directory: $Username"}
}
```

Then you use the `Else` part to say that if the account was not already found let's create it. So the `New-ADUser` cmdlet is called with the various parameters that were captured in the CSV file.

```
else
{
    New-ADUser `
        -SamAccountName $Username `
        -UserPrincipalName "$Username@sometestorg.com" `
        -Name "$FName $LName" `
        -GivenName $FName `
        -Surname $LName `
        -Enabled $True `
        -DisplayName "$LName, $FName" `
        -Path $OU `
        -OfficePhone $Phone `
        -EmailAddress $Email `
        -Title $Title `
        -Department $Dept `
        -AccountPassword (convertto-securestring $Password
        -AsPlainText -Force) -ChangePasswordAtLogon $True
}
```

Those are the separate pieces, so let's see the script in its entirety. *Note:* I've removed the comments from the example to make the print version more readable. The version on GitHub has all of the comments:

```
Import-Module ActiveDirectory
$ImportADUsers = Import-Csv C:\PSTemp\UserImport.csv

foreach ($User in $ImportADUsers)
{
    $FName = $User.FName
    $LName = $User.LName
    $Username = $User.username
    $Email = $User.Email
    $Phone = $User.Phone
    $Dept = $User.Dept
    $Password = $User.password
    $Title = $User.Title
    $OU = $User.OU
```



```

if (Get-ADUser -Filter {SamAccountName -eq $Username})
{
    Write-Warning "This user account already exists in Active
Directory: $Username"
}
else
{
    New-ADUser `
        -SamAccountName $Username `
        -UserPrincipalName "$Username@sometestorg.com" `
        -Name "$FName $LName" `
        -GivenName $FName `
        -Surname $LName `
        -Enabled $True `
        -DisplayName "$LName, $FName" `
        -Path $OU `
        -OfficePhone $Phone `
        -EmailAddress $Email `
        -Title $Title `
        -Department $Dept `
        -AccountPassword (convertto-securestring $Password
-AsPlainText -Force) -ChangePasswordAtLogon $True

    }
}

```

Running the script

The first time you run the script, I highly recommend that you run it within Visual Studio Code. Visual Studio Code is helpful in troubleshooting issues with scripts because you can do debugging and execute one line of code at a time. Plus, it highlights issues for you, which can make them easier to find. The usual culprits are those darn braces on the loops.

To open and run in PowerShell ISE, follow these steps:

1. Click Start, scroll down to Visual Studio Code, expand the folder and click Visual Studio Code.
2. Choose File ⇨ Open File.
3. Navigate to your script.

4. Select the script and click Open.
5. To run the script, choose Terminal ⇨ Run Active File

When you're sure that your script is working, there are two methods to run it:

- » **Right-click the script and select Run with PowerShell.** I don't like this method because the PowerShell window pops up but closes right away when you're done, so you can't see if it encountered any errors.
- » **Open a PowerShell window and run it by specifying the directory (Example 1) or running it from the same directory (Example 2):**
 - **Example 1:** `C:\PSTemp\UserImport.ps1`
 - **Example 2:** `.\UserImport.ps1`

Defining a Script Policy

Defining a script execution policy allows you to define what kind of scripts are allowed to run within your network. You can set the execution policy through Group Policy organization wide, or through the following PowerShell cmdlet. The execution policy is set to Restricted by default.

```
Set-ExecutionPolicy -ExecutionPolicy <policy>
```

Here are the policy types that can be used to set execution policy:

- » **Restricted:** Prevents PowerShell scripts from running and will not load configuration files.
- » **AllSigned:** For a script to run, it must be signed by a trusted certificate.
- » **RemoteSigned:** Requires that any script that is downloaded from the Internet be signed by a trusted certificate. Scripts created locally do not have to be signed to run.
- » **Unrestricted:** Allows you to run all scripts. You're prompted for permission before a script is run.

- » **Bypass:** Similar to Unrestricted, but it doesn't prompt for permission to run.
- » **Undefined:** Removes whatever execution policy is currently set, unless that execution policy is being set through Group Policy.

Signing a PowerShell Script

Depending on how your execution policy is set, you may be able to run scripts that you've created without any issue. If you're in a more secure environment, however, you may need to sign your script so that it will be trusted and allowed to run.

Check out Book 6, Chapter 1 for more on code signing. There, I walk you through the steps of requesting a code signing certificate and signing a PowerShell script.

Creating a PowerShell Advanced Function

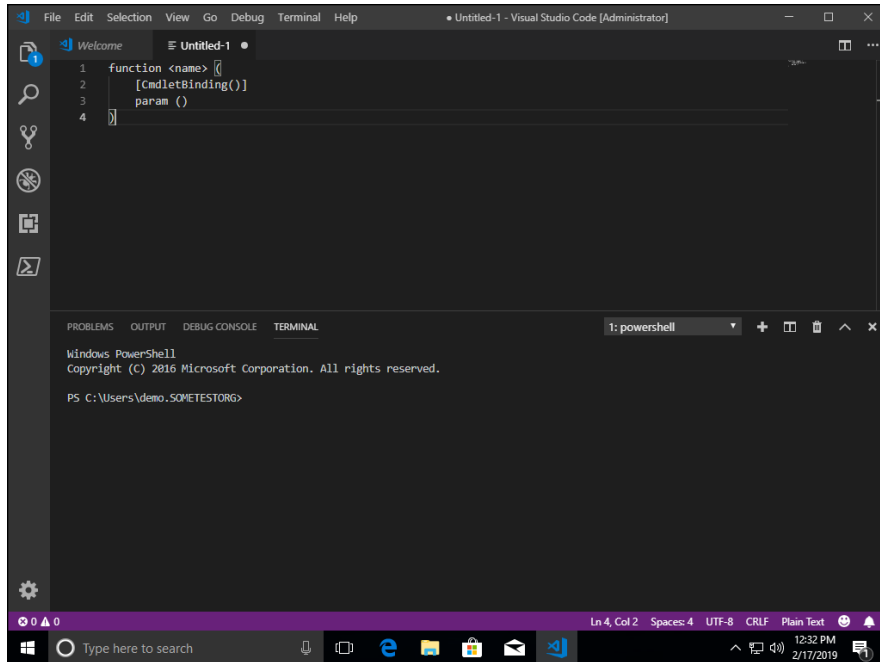
For most system administrators, the idea of making your own tools in PowerShell can be a little intimidating. With PowerShell advanced functions, you can use much of what you've learned about PowerShell to create your own tool set that you can run just like you run PowerShell cmdlets. The biggest difference is that PowerShell cmdlets are written in .NET Framework languages like C#, and you must compile them to use them. Advanced functions are written using the PowerShell scripting language.

There are a few components that go into creating a PowerShell advanced function. I'll cover these components first, before I dig into creating your first advanced function.

- » `[CmdletBinding()]`: This is what changes a function into an advanced function. It not only allows the function to operate like a cmdlet, but also allows you to use cmdlet features.
- » `param`: This area is used to set the parameters that you want your advanced function to use.

You can see how these components are laid out in Figure 4-3.

FIGURE 4-3:
The basic anatomy of an advanced function includes `[CmdletBinding()]`, which allows the function to behave like a cmdlet and use cmdlet features.



Playing with parameters

Advanced functions give you a lot of granularity when it comes to working with parameters that you just don't have with basic functions. These are placed in the parameter block where you define the parameters for your function. Here are a few of my favorites:

- » **Mandatory parameters:** When you specify a parameter as mandatory, the function will not be able to run if that parameter is not provided. In the following example, I've set the parameter to be mandatory, and I've indicated that the value for the parameter will come from the pipeline.

```
[Parameter(Mandatory,ValueFromPipeline)]
```

- » **Parameter validation:** Parameter validation is very useful when you want to ensure that a parameter matches some form of expected input. This is done by typing **ValidateSet**, and then by specifying the strings you expect to see. If the parameter string does not match, then the function will not be able to run. See the following example:

```
[ValidateSet('String1','String2')]
```

Creating the advanced function

Now that you know the basic building blocks of advanced functions, I'll create an example of an advanced function. This advanced function will retrieve information about a system. I'll start with the individual components of the function, and then I'll show you the whole thing after it's done. The function is also available for download from this book's GitHub repository at <https://github.com/sara-perrott/Server2019PowerShell>.

First, to tell PowerShell that you want to create a function, you need to start the text in the file with `function`, followed by what you want to name your function. This absolutely has to follow the PowerShell syntax of `verb-noun`. Next up, you add `[CmdletBinding()]`, which tells PowerShell that this is an advanced function and should be treated similarly to a cmdlet. In the `Param` block, you define any parameters you want to use. In this case, you're defining one parameter, which is a variable named `computername`.

```
function Get-ReconData
{
    [CmdletBinding()]
    Param (
        [string[]] $computername
    )
}
```

Next, I'll add some text in a `BEGIN` block. I like to use this to see which system it's on currently. You probably won't want to do this in production, especially if you have multiple systems that you're running this function against. But it's great for troubleshooting issues in your script.

```
BEGIN {
    Write-Output "Gathering reconnaissance on $computername"
}
```

Next up is where the magic happens. The `PROCESS` block is where I'm telling the function what I want it to do. In this example, I'm telling it to run the code block for every object that is passed to it through the `$computername` variable. Each object is assigned to the `$computer` variable.

Now you can make use of Windows Management Instrumentation (WMI) classes to get the information that you want. In this case, I'm using two WMI classes to query for the data that I want. I'm using the `Win32_OperatingSystem` class and the `Win32_ComputerSystem` class. You can see a listing of the properties that you can work with on the Microsoft documentation pages. `Win32_OperatingSystem` can be found at <https://docs.microsoft.com/en-us/windows/desktop/cimwin32prov/win32-operatingsystem> and `Win32_ComputerSystem` can be found at <https://docs.microsoft.com/en-us/windows/desktop/cimwin32prov/win32-computersystem>.

To tell PowerShell that I want to use the two WMI classes, I use the PowerShell cmdlet `Get-WmiObject` to assign the desired WMI class to a variable. I then chose a few of the properties that I felt were most useful for gathering some information about the system. I created names for them, and then mapped the name to the WMI variable I created earlier and the property that I'm interested in. Finally, I'm telling it to write the output of the function to the screen. In a production environment, if you were running this against multiple systems, you could export the data to a file.

```
PROCESS {
    foreach ($computer in $computername) {
        $os = Get-WmiObject -class Win32_OperatingSystem
        -computerName $computer
        $comp = Get-WmiObject -class Win32_ComputerSystem
        -computerName $computer
        $prop = @{ 'ComputerName'=$computer;
                  'OSVersion'=$os.version;
                  'SPVersion'=$os.servicepackmajorversion;
                  'FreeMem'=$os.FreePhysicalMemory;
                  'OSType'=$os.OSType;
                  'Domain'=$comp.domain;
                  'Status'=$comp.Status}
        $sysinfo = New-Object -TypeName PSObject -Property $prop
        Write-Output $sysinfo}
    }
```

The last block is simply the `End{}` block. In this case, you don't need it to run anything after the function has ran, so it's left blank.

Here is the final function in all its glory!

```
function Get-ReconData
{
    [CmdletBinding()]
    Param (
        [string[]] $computername
    )
    BEGIN {
        Write-Output "Gathering reconnaissance on $computername"
    }
    PROCESS {
        foreach ($computer in $computername) {
            $os = Get-WmiObject -class Win32_OperatingSystem
            -computerName $computer
```

```

        $comp = Get-WmiObject -class Win32_ComputerSystem
        -computerName $computer
        $prop = @{ 'ComputerName'=$computer;
                    'OSVersion'=$os.version;
                    'SPVersion'=$os.servicepackmajorversion;
                    'FreeMem'=$os.FreePhysicalMemory;
                    'OSType'=$os.OSType;
                    'Domain'=$comp.domain;
                    'Status'=$comp.Status}
        $sysinfo = New-Object -TypeName PSObject -Property
        $prop
        Write-Output $sysinfo}
    }
END {}
}

```

Save your function as a .ps1 file just as you would a normal PowerShell script. Now let's try it and see what it looks like when it is run.

Using the advanced function

You can run functions from your code editor of choice or from PowerShell. They're all a little different as to how you should execute the code. In this section, I cover running the advanced function in VS Code, which you'll most likely do while testing, and running it in PowerShell, which is the more realistic production method.

Running an advanced function in VS Code

Now that you've written the advanced function, you want to try it out and see if it works. I'll use VS Code to test it. This is very common when wanting to validate that your function is working properly. I'll use the names and parameters from my previous example.

1. Click the Start menu and scroll down to the Visual Studio Code folder.
2. Expand the folder and click Visual Studio Code to launch it.
3. Choose File ⇨ Open File.
4. Navigate to where your script is stored, select the file, and click Open.
5. In the Terminal, navigate to where you save your function.

In my case this is Documents, so I typed **cd Documents**.

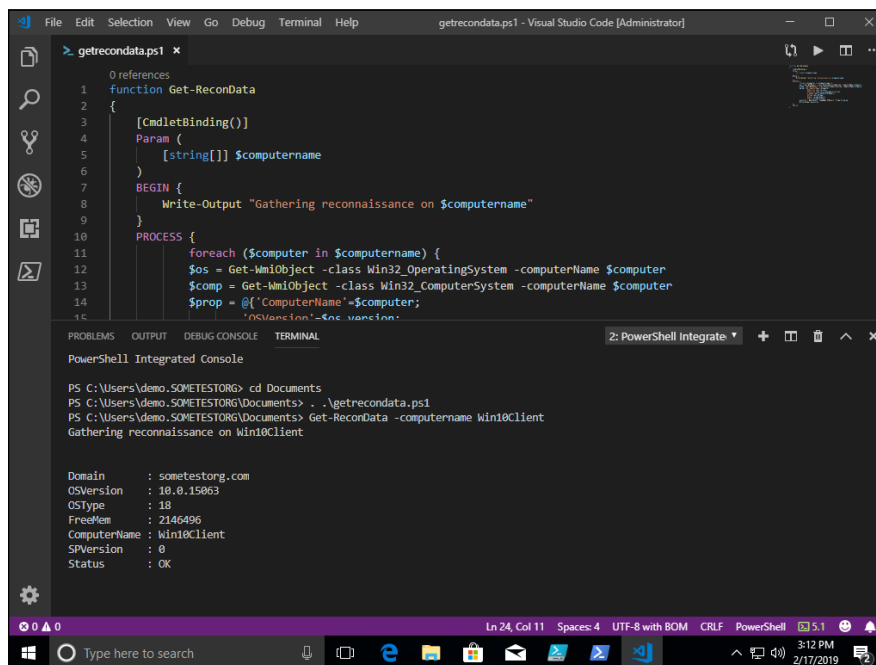
6. Type a period, followed by a space, and then `.\getrecondata.ps1`.

This tells it that you want to run the script from the current directory.

7. Type `Get-ReconData -computername Win10Client`.

The output you receive will contain all the information you asked the function to retrieve. In this case, `Win10Client` is a Windows 10 virtual machine, but you can see that I'm still able to get valuable information, shown in Figure 4-4.

FIGURE 4-4:
Testing your
function is
important to
do, and it's
simple from
within VS Code.



```
0 references
1 function Get-ReconData
2 {
3     [CmdletBinding()]
4     Param (
5         [string[]] $computername
6     )
7     BEGIN {
8         Write-Output "Gathering reconnaissance on $computername"
9     }
10    PROCESS {
11        foreach ($computer in $computername) {
12            $os = Get-WmiObject -class Win32_OperatingSystem -computerName $computer
13            $comp = Get-WmiObject -class Win32_ComputerSystem -computerName $computer
14            $prop = @('ComputerName' -> $computer;
15                    'OSVersion' -> $os.Version;
```

PowerShell Integrated Console

```
PS C:\Users\demo.SOMETESTORG> cd Documents
PS C:\Users\demo.SOMETESTORG\Documents> .\getrecondata.ps1
PS C:\Users\demo.SOMETESTORG\Documents> Get-ReconData -computername Win10Client
Gathering reconnaissance on Win10Client

Domain       : sometestorg.com
OSVersion    : 10.0.15063
OSType       : 18
FreeSpace    : 2146496
ComputerName : win10client
SPVersion    : 0
Status       : OK
```

Running an advanced function in PowerShell

In a production environment, it's far more likely that you'll choose to run the advanced function from PowerShell rather than a code editor. Here's how to do this:

1. **Right-click the Start menu and choose Windows PowerShell (Admin).**
2. **Navigate to the location where your function is saved.**

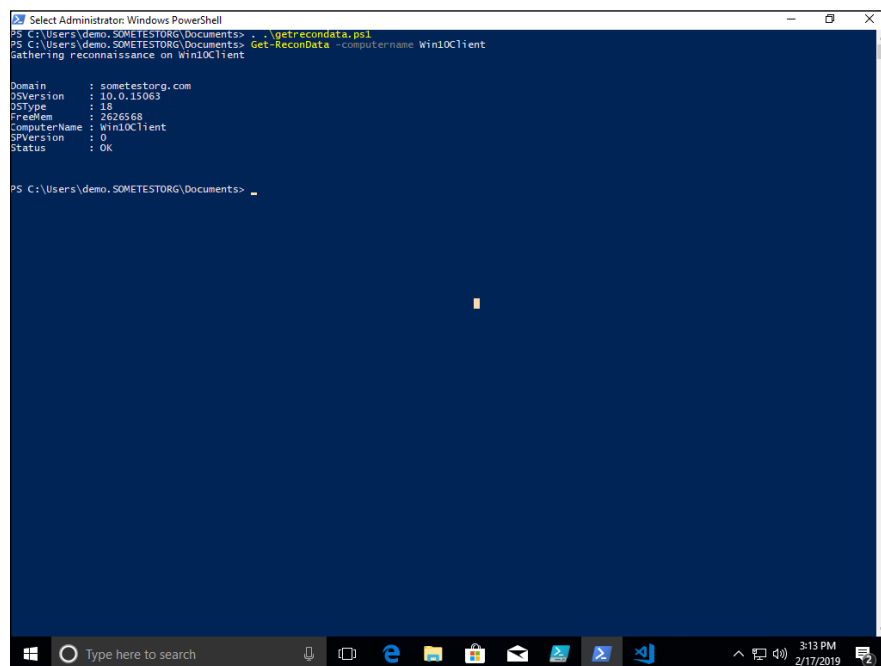
In my case, it's in my Documents folder, so I typed `cd Documents` to get to it.

3. Type a period, followed by a space, and then `.getrecondata.ps1`.

This tells it that you want to run the script from the current directory.

4. Type `Get-ReconData -computername Win10Client`.

After the function has run, the information you requested is output to the screen, as shown in Figure 4-5.



```
Select Administrator: Windows PowerShell
PS C:\Users\demo.SOMETESTORG\Documents> . .getrecondata.ps1
PS C:\Users\demo.SOMETESTORG\Documents> Get-ReconData -computername Win10Client
gathering reconnaissance on win10client

Domain       : sometestorg.com
OSVersion    : 10.0.15063
OSType       : 18
FreeSpace    : 2626568
ComputerName : win10client
SPVersion    : 0
Status       : OK

PS C:\Users\demo.SOMETESTORG\Documents>
```

FIGURE 4-5:
You can run
your advanced
function in your
PowerShell
console.

IN THIS CHAPTER

- » Introducing PowerShell Desired State Configuration
- » Making your first PowerShell Desired State Configuration script
- » Using the PowerShell Desired State Configuration script
- » Implementing PowerShell Desired State Configuration at scale

Chapter 5

PowerShell Desired State Configuration

PowerShell Desired State Configuration (DSC) enables system administrators to configure systems and keep them in compliance with set organizational baselines. It's sometimes referred to Configuration as Code.

In this chapter, I introduce you to PowerShell DSC and tell you how to create a DSC script and how to apply it.

Getting an Overview of PowerShell Desired State Configuration

DSC was introduced in PowerShell version 4. It provided a simple way to specify what you wanted a system to be, instead of having to build a system one line of code at a time. I like to compare it to baking: Traditional PowerShell is like following the recipe; DSC is like saying, “I want this,” and handing the picture of a cake to the baker.

DSC is exceptionally powerful. It can be used to install roles and features, copy files to specified locations, install software, and make changes to the Registry.

PowerShell DSC has three main components:

- » **Configurations:** PowerShell scripts that are used to configure your resources according to your organization's requirements
- » **Resources:** Code that keeps your system in compliance with the specified configuration
- » **Local Configuration Manager (LCM):** Handles the interactions between the configurations and the resources

In the following sections, I cover each of these components in a little deeper detail and show you what they look like in an actual DSC script.

Configurations

Because PowerShell DSC files are saved as PS1 files, you may be wondering how PowerShell knows that the file is a PowerShell DSC file. That's a great question! It looks for the keyword *configuration*. This tells it that the file contains a DSC configuration. A sample in a DSC configuration may look something like this:

```
Configuration MyAwesomeWebsite {
```



TIP

You can name the configuration whatever you like. PowerShell doesn't care what it's called. I suggest giving it a name that makes sense to you so that you can look at it and know from the name what you're configuring with this particular configuration file.

A DSC script accomplishes its tasks with a slightly different feel from traditional PowerShell; it looks similar to an advanced function. With traditional PowerShell, you would write something along the lines of the following:

```
Install-WindowsFeature -Name "Web-Server"
```

This installs the Internet Information Services (IIS) web server on the Windows system that you run it on. You may also specify whether you want sub-features to be enabled and if you want the management tools for IIS to be installed.

With a DSC configuration, this is how you would accomplish the same thing as that line of PowerShell:

```
WindowsFeature WebServer {  
    Ensure = "Present"  
    Name    = "Web-Server"  
}
```

Instead of telling it explicitly to install the `Web-Server` feature via PowerShell, you simply tell DSC that you want to make sure that `Web-Server` is present. If it is, then the script continues to execute and does not reinstall the `Web-Server`. If it is not installed, then the `Web-Server` feature will be installed.

Resources

Resources are the foundational pieces of PowerShell DSC. Resources make properties available that can contain PowerShell scripts and that can be used by LCM to implement the changes.

You may want to look up how to use a particular DSC resource. This is simple to do with the following command:

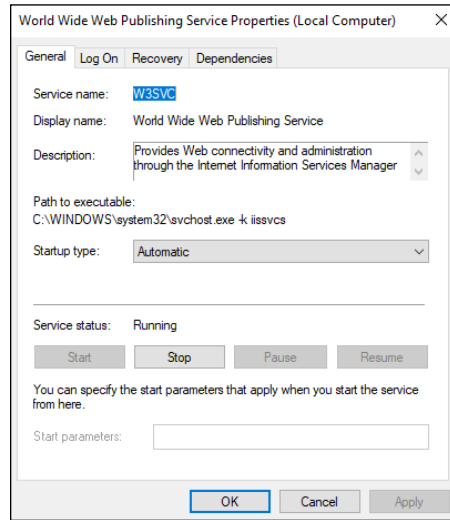
```
Get-DscResource Syntax <resource>
```

If, for instance, you run that command using `service` as the resource, you'll find some of the more common things that are used in relation to services, like the service name and the state of the service. Most administrators, for example, will want to check to see if a service is running. Maybe you want to verify that your antivirus service or patching service is running. The following example shows you how you can use a resource block to verify that a service is running on your local system, in this case IIS:

```
Configuration MyAwesomeWebsite  
{  
    Import-DSCResource -Name Service  
    Node localhost  
    {  
        Service "W3SVC:Running"  
        {  
            Name = "W3SVC"  
            State = "Running"  
        }  
    }  
}
```

You may be wondering how I got that service name for IIS, the web server available in Microsoft Server 2019. You can go into the `Services.msc` panel and get your service names there. In the example of the web server, I scrolled down to World Wide Web Publishing Service and double-clicked it. The name that you need for DSC is listed as Service Name, shown in Figure 5-1 as `W3SVC`.

FIGURE 5-1:
The service name is displayed within the `services.msc` panel for each service.



Of course, `Service` is not the only resource type available. The `Import-DSCResource` module contains many different resources that you can use in your DSC scripts. I have listed the more common resource types in Table 5-1.

TABLE 5-1 **DSC Resources**

Resource Name	Description
file	The file resource type can be used to copy files from source to destination and ensure that the files in the source and destination always match. It can use dates and hashes to compare the source and destination files. If they don't match, the source files are copied over the destination files.
archive	The archive resource type can unpack archive files like ZIP files. It can validate the integrity of the archive file against a checksum.
environment	The environment resource type can be used to work with environmental variables that you want to create, modify, or manage.
group	The group resource type can be used to work with groups. This includes the management of the groups and the users within the groups.

Resource Name	Description
log	The log resource type is used for exactly what you may think: It writes to an event log. Specifically it writes to the Microsoft-Windows-Desired State Configuration, Analytic event log.
package	The package resource type is very useful when you want to ensure that something is installed. You can also use it to uninstall packages.
registry	The registry resource type allows you to work with the Registry, including the creation, modification, and deletion of Registry keys and their associated values.
script	The script resource type allows you to have a little fun and create your own script blocks. Your script may start with Get, Set, or Test.
service	The service resource type allows you to work with services that are on the system. You can ensure that they're present and that they're running or disabled.
user	The user resource type allows you to work with local user accounts on the system. You can create, modify, and delete users, as well as make changes to their accounts.
WindowsFeature	The WindowsFeature resource type is one of the more common types that you see if you're in a shop that is doing Infrastructure as Code. It can be used to work with both roles and features, and can install, uninstall, or modify both roles and features.
WindowsProcess	The WindowsProcess resource type allows you to work with windows processes. It allows you to start, stop, and configure them.

If this table doesn't list a resource you need, you can download additional resources from the PowerShell Gallery (www.powershellgallery.com) and GitHub (<https://github.com/PowerShell/DscResources>).

Local Configuration Manager

The LCM receives configurations that are sent to your systems and applies those configurations. It runs on every system that you're targeting with your DSC scripts.

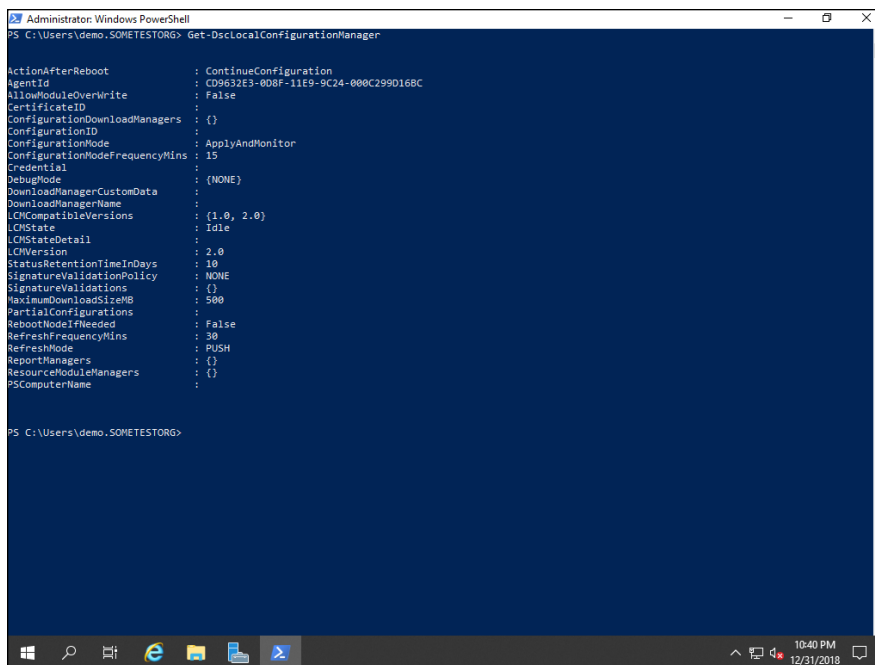
LCM allows you to specify how you want to get configurations. You can use a push model or a pull model. You can also set how often each system reaches out to pull a configuration down.

If you want to see that the LCM settings are on a system on which you want to use DSC configuration scripts, you can run the following command:

```
Get-DscLocalConfigurationManager
```

This command displays the current settings for LCM. An example is shown in Figure 5-2.

FIGURE 5-2:
Checking the
current settings
for LCM
is simple to do in
PowerShell
with the Get-
DscLocal
Configuration
Manager
command.



```
Administrator: Windows PowerShell
PS C:\Users\demo\SOMETESTORG> Get-DscLocalConfigurationManager

ActionAfterReboot      : ContinueConfiguration
AgentId                : CD9632f3-008f-11e9-9c24-000c299d168c
AllowModuleOverWrite   : False
CertificateId          : 
ConfigurationDownloadManagers : {}
ConfigurationId        : 
ConfigurationMode      : ApplyAndMonitor
ConfigurationModeFrequencyMins : 15
Credential             : 
DebugMode              : (NONE)
DownloadManagerCustomData : 
DownloadManagerName    : 
LCMCompatibleVersions  : {1.0, 2.0}
LCMState               : Idle
LCMStateDetail         : 
LCMVersion             : 2.0
StatusRetentionTimeInDays : 10
SignatureValidationPolicy : NONE
SignatureValidations    : {}
MaximumDownloadSizeMB : 500
PartialConfigurations  : 
RebootModeIfNeeded     : False
RefreshFrequencyMins   : 30
RefreshMode            : PUSH
ReportManagers         : {}
ResourceModuleManagers : {}
PCComputerName         : 

PS C:\Users\demo\SOMETESTORG>
```

Creating a PowerShell Desired State Configuration Script

If you've read this chapter from the beginning, you've found out about the components of DSC at this point. Now how do you build something useful? In this section, you continue to build on the example of a web server. In your organization, you may need to be able to provision systems quickly to keep up with demand. In this example, you're going to use DSC to ensure that IIS is installed on the system and to copy files for the website over from a source that is specified to the new web server so that it can start serving out content. Normally, the source would not be on the same server; it would simply be available on the same network.

The script starts with the keyword `configuration`, which tells PowerShell this is a DSC file. The next step is to import `PSDesiredConfiguration`, which is needed to load the custom resources that you may need in your script. `Node` is used to specify the system that you want to run the DSC against. In this case, I'm running it on the computer that it resides on, so `Node` is set to `localhost` (my local computer). The `WindowsFeature` block is where I tell it that the `WebServer` feature must be present. This will check to see if Internet Information Services (IIS) is installed. If it is not installed, then DSC will install the feature. The last part of this script copies the website files from the source to the destination. This can be a great way to speed the provisioning of web servers if you need to scale quickly.


```

Configuration MyAwesomeWebsite {

    Import-DscResource -ModuleName PsDesiredStateConfiguration

    Node 'localhost' {
        WindowsFeature WebServer {
            Ensure = "Present"
            Name    = "Web-Server"
        }
        File WebsiteGoodies {
            Ensure = 'Present'
            SourcePath = 'c:\PStemp\index.html'
            DestinationPath = 'c:\inetpub\wwwroot'
        }
    }
}
MyAwesomeWebsite

```

You'll want to write your script in your code editor of choice and save it to a PS1 file. I've made this simple DSC script available on the GitHub for this book. You can reach it at <https://github.com/sara-perrott/Server2019PowerShell>.

You may notice that this DSC configuration was written with just one node, localhost. In most production instances, you're going to want to target multiple hosts with your DSC configuration. This is a simple change to make. Instead of the single node being defined like this:

```
Node 'localhost' {
```

It gets defined like this to support multiple nodes:

```
Node @('localhost','Server2') {
```

Applying the PowerShell Desired State Configuration Script

After you have your script written, you'll want to actually apply it. The steps are fairly simple. First, run the PS1 file as you would any other PowerShell script. This will create a Managed Object Format (MOF) file. MOF files are what actually get used by PowerShell DSC to do its configuration work.

Compiling into MOF

Open your PowerShell window. You can reach this by right-clicking Start and then choosing Windows PowerShell Admin. Then follow these steps:

1. **Navigate to the directory where the new PS1 file is stored.**
2. **Run the following command:**

```
.. \AwesomeWebsite.ps1
```

You can see in Figure 5-3 that running the command generates the MOF file. The MOF file takes its name from the name supplied in the resource block. In this case, the name is `localhost`. If you have multiple systems, you'll have multiple MOF files, one for each system represented.

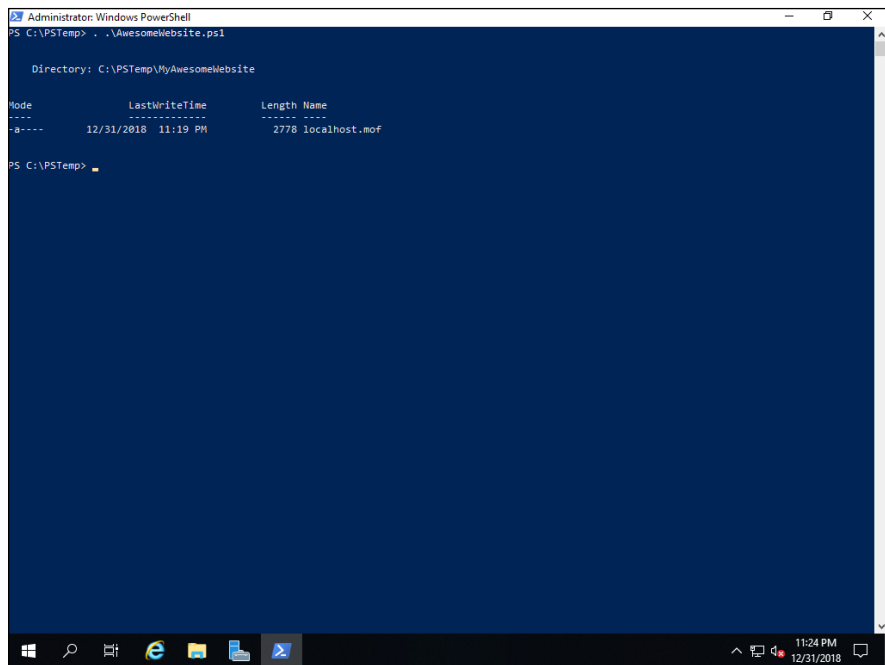


FIGURE 5-3:
Compiling the PS1 into an MOF file is done by running the PowerShell script as you normally would.

If you browse to the directory where your script is located, you'll notice that you now have a folder with the name of your configuration on it. Inside of this folder is your MOF file.

Applying the new configuration

Now that you've created the MOF file, it's time to apply the configurations contained within it. You'll want to run the `Start-DscConfiguration` cmdlet, which will use the MOF file to run the configuration against the system. You run the cmdlet and specify the folder that the MOF file is in:

```
Start-DscConfiguration .\MyAwesomeWebsite
```

After this command is run, you get the state of the configuration, which should be **Running**, as shown in Figure 5-4.

```

PS C:\PSTemp> .\AwesomeWebsite.ps1

Directory: C:\PSTemp\MyAwesomeWebsite

Mode                LastWriteTime         Length Name
----                -
-a-----         12/31/2018 11:29 PM             2778 localhost.mof

PS C:\PSTemp> dir

Directory: C:\PSTemp

Mode                LastWriteTime         Length Name
----                -
d-----         12/31/2018 11:29 PM             MyAwesomeWebsite
-a-----         12/31/2018 11:19 PM             444 AwesomeWebsite.ps1
-a-----         12/31/2018 11:03 PM             94 index.html

PS C:\PSTemp> Start-DscConfiguration .\MyAwesomeWebsite\

Id  Name      PSJobTypeName State      HasMoreData Location      Command
--  -
2   Job2      Configuratio... Running    True        localhost     Start-DscConfiguration...

PS C:\PSTemp>
  
```

FIGURE 5-4:
The DSC script
has been
compiled and
has been run;
you can see the
State is currently
Running.

Push and Pull: Using PowerShell Desired State Configuration at Scale

As you can see, DSC is a very powerful tool for configuring your systems. Chances are, you want to automate things a bit more. You don't want to log on to your server and run DSC manually. That's where you need to start looking at the two configuration modes: push mode and pull mode.

Push mode

In push mode, the configuration is pushed to the destination system. It's a one-way relationship. Push mode is done similar to the example in the “Applying the new configuration” section earlier in this chapter. You run the `Start-DscConfiguration` cmdlet, and you can use the `-ComputerName` parameter to target specific systems with the script.



TIP

You can easily automate this on a scripting server with scheduled tasks. By setting it up as a scheduled task, you can ensure that your configuration is pushed at regular intervals and that your systems are configured exactly the way you expect them to be configured.

When using push mode, you have a decision to make when it comes to the configuration mode that you want to use. You can see which one you're using currently by running the `Get-DscLocalConfigurationManager` command. There are three options:

- » `ApplyOnly`: Applies the configuration once, but does nothing further.
- » `ApplyandMonitor`: Applies the configuration and will write any discrepancies to the logs. This is the default value.
- » `ApplyandAutoCorrect`: Applies the configuration, writes discrepancies to the logs, and then applies the current configuration again.

If you make a change to the configuration script, you need to regenerate the MOF. DSC does cache configurations, so if you have made changes to your script, you need to stop and restart the process that is hosting the DSC engine. You can do that with the following commands.

First, you need to get the process ID of the process that is hosting the DSC engine.

```
$ProcessID = Get-WmiObject msft_providers | Where-Object {$_.  
    provider -like 'dsccore'} | Select-Object -ExpandProperty  
    HostProcessIdentifier
```

Then you stop the process using the following command:

```
Get-Process -Id $ProcessID | Stop-Process
```

After you've stopped the process, simply run `Start-DscConfiguration` again, and it will use the newest version of the configuration script.

Pull mode

Pull mode is a little more complicated to get started. You need to set up a pull server, which will host the DSC service and contain all the configuration and resource scripts that the clients will pull.

The sequence for setting up pull mode is to create the configuration script, set up the pull server, and then set up DSC on the system that you want the pull to occur from.

Setting up the pull server

Before you can use the pull server, you need to configure it. First, you need to install a module named `xPSDesiredStateConfiguration`. This module is available on PowerShell Gallery. With PowerShell open, you can run the command `Install-Module -Name xPSDesiredStateConfiguration`, which automatically downloads and installs the module for you. You may be prompted to accept a new provider named Nuget. Press Y if that occurs. You'll be asked if you're sure you want to install the module; press Y.

After the module is installed, you can run the script to set up the DSC pull server. I'm using the configuration script available from Microsoft's DSC pull page. You can copy the script from <https://docs.microsoft.com/en-us/powershell/dsc/pull-server/pullServer>.

Configuring DSC on the system to use the pull server

After the pull server is set up, the last step is to set up the system that will be pulling the configuration from the pull server. You can run the following to set up the client.

```
Configuration ConfigurationForPull
{
    LocalConfigurationManager
    {
        ConfigurationID =
        "registration_key_from_server_setup_script";
        RefreshMode = "PULL";
        DownloadManagerName = "WebDownloadManager";
        RebootNodeIfNeeded = $true;
        RefreshFrequencyMins = 30;
```

```
        ConfigurationModeFrequencyMins = 60;
        ConfigurationMode = "ApplyAndAutoCorrect";
        DownloadManagerCustomData = @{ServerUrl = "http://
PullServer:8080/PSDSCPullServer/PSDSCPullServer.svc";
        AllowUnsecureConnection = "TRUE"}
    }
}
ConfigurationForPull -Output "."
```

Save the script and run it like you have before. And then use the `Start-DscConfiguration` cmdlet to apply the configuration.