5

Exploring .NET

In this chapter, we cover the following recipes:

- Exploring .NET assemblies
- Examining .NET classes
- Leveraging .NET methods
- Creating a C# extension
- Creating a PowerShell cmdlet

Introduction

Microsoft first launched the Microsoft .NET Framework in June 2000, amid a frenzy of marketing zeal, with the code name Next Generation Windows Services. Microsoft seemed to add the .NET moniker to every product. There was Windows .NET Server (renamed Windows Server 2003), Visual Studio .NET, and even MapPoint .NET. As is often the case, over time, .NET provided features which were superseded by later and newer features based on advances in technology. For example, **Simple Object Access Protocol** (**SOAP**) and XML-based web services have given way to **Representation State Transfer** (**REST**) and **JavaScript Object Notation** (**JSON**).

Microsoft made considerable improvements to .NET with each release and added new features in response to customer feedback. .NET started as closed source as the .NET Framework. Microsoft then transitioned .NET to open source, aka .NET Core. PowerShell 7.0 is based on .NET Core 3.1.

An issue was that, over time, .NET became fragmented across different OSes and the web. To resolve this, Microsoft created .NET 5.0 and dropped the "Core" moniker. The intention going forward is to move to a single .NET across all platforms and form factors, thus simplifying application development. See `https://www.zdnet.com/article/microsoft-takes-a-major-step-toward-net-unification-with-net-5-0-release/` for some more details on this.

It is important to note that neither .NET 5.0 nor PowerShell 7.1 have **long-term support** (**LTS**). The next LTS version of .NET is .NET 6.0 and of PowerShell is PowerShell 7.2, both of which are not meant to be released until late in 2021. PowerShell 7.1 is built on top of, and takes advantage of, the latest version of .NET Core, now known as .NET 5.0.

For the application developer, .NET is primarily an **Application Program Interface** (**API**) and a programming model with associated tools and runtime. .NET is an environment in which developers can develop rich applications and websites across multiple platforms.

For IT professionals, .NET provides the underpinnings of PowerShell but is otherwise transparent. You use cmdlets to carry out actions. These cmdlets use .NET to carry out their work without the user being aware there is .NET involved. You can use `Get-Process` without worrying how that cmdlet actually does its work. But that only gets you so far; there are some important .NET features that have no cmdlets. For example, creating an Active Directory cross-forest trust is done by using the appropriate .NET classes and methods (and is very simple too!). .NET is an important depth tool, helping you to go beyond the cmdlets provided in Windows and Windows applications.

Here is a high-level illustration of the components of .NET:

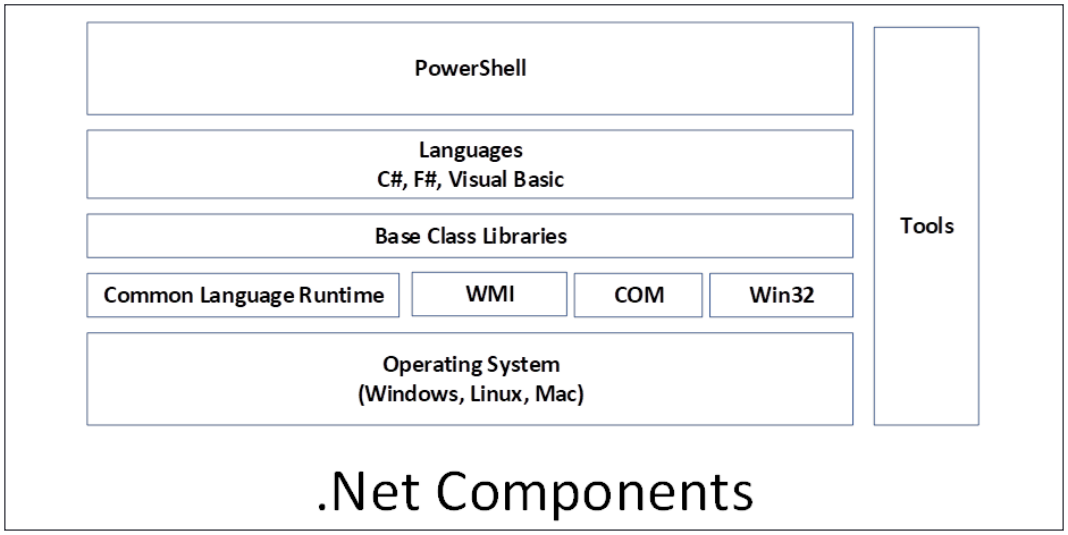

Figure 5.1: .NET components

The key components of .NET in this diagram are:

- **Operating System**: The .NET story begins with the operating system. The operating system provides the fundamental operations of your computer. .NET leverages the OS components. The .NET team supports .NET 5.0, the basis for PowerShell, on Windows, Linux, and Apple Mac. .NET also runs on the ARM platform, which enables you to get PowerShell on ARM. For a full list of operating systems that support PowerShell 7, see `https://github.com/dotnet/core/blob/master/release-notes/5.0/5.0-supported-os.md`.
- **Common Language Runtime** (**CLR**): The CLR is the core of .NET, the managed code environment in which all .NET applications run (including PowerShell). The CLR manages memory, threads, objects, resource allocation/de-allocation, and garbage collection. For an overview of the CLR, see `https://docs.microsoft.com/dotnet/standard/clr/`. For PowerShell users, the CLR "just works."
- **Base Class Libraries** (**BCLs**): .NET comes with a large number of base class libraries, the fundamental built-in features that developers use to build .NET applications. Each BCL is a DLL which contains .NET classes and types. When you install PowerShell, the installer adds the .NET Framework components into PowerShell's installation folder. PowerShell developers use these libraries to implement PowerShell cmdlets. You can also call classes in the BCL directly from within PowerShell. It is the BCLs that enable you to reach into .NET.
- **WMI, COM, Win32**: These are traditional application development technologies which you can access via PowerShell. Within PowerShell, you can run programs that use these technologies, as well as access them from the console or via a script. Your scripts can be a mixture of cmdlets, .NET, WMI, COM, and Win32 if necessary.
- **Languages**: This is the language that a cmdlet and application developer uses to develop PowerShell cmdlets. You can use an extensive variety of languages, based on your preferences and skill set. Most cmdlet developers use C#, although using any .NET supported language would work, such as VB.Net. The original PowerShell development team designed the PowerShell language based on C#. You can describe PowerShell as on the glide scope down to C#. This is useful since there is a lot of documentation with examples in C#.
- **PowerShell**: PowerShell sits on top of these other components. From PowerShell, you can use cmdlets developed using a supported language, and you can use both BCL and WMI/COM/Win32 as well.

For the most part, moving to .NET 5 with respect to PowerShell is more or less transparent, although not totally. One thing that's changed in .NET 5 is it drops support for a lot of WinRT APIs. The .NET team did this to boost cross-platform support. However, this means that any module that relied on these APIs, such as the Appx module (`https://github.com/PowerShell/PowerShell/issues/13138`), is no longer compatible in PowerShell 7.1 (when they were in PS7), although they can be used via the Windows PowerShell compatibility solution described in *Chapter 3*, *Exploring Compatibility with Windows PowerShell*.

In this chapter, you examine the assemblies that make up PowerShell 7.1. You then look at both the classes provided by .NET and how to leverage the methods provided by .NET. You also look at creating a simple C# extension and creating a PowerShell cmdlet.

Exploring .NET assemblies

With .NET, an assembly holds compiled code which .NET can run. An assembly can either be a **Dynamic Link Library** (**DLL**) or an executable. Cmdlets and .NET classes are contained in DLLs, as you see in this recipe. Each assembly also contains a manifest which describes what is in the assembly, along with compiled code.

Most PowerShell modules and commands make use of assemblies of compiled code. When PowerShell loads any module, the module manifest (the `.PSD1` file) lists the assemblies which make up the module. For example, the `Microsoft.PowerShell.Management` module provides many core PowerShell commands, such as `Get-ChildItem` and `Get-Process`. This module's manifest lists a nested module (that is, `Microsoft.PowerShell.Commands.Management.dll`) as the assembly containing the actual commands.

A great feature of PowerShell is the ability for you to invoke a .NET class method directly or to obtain a static .NET class value. The syntax for calling a .NET method or a .NET field, demonstrated in numerous recipes in this book, is to enclose the class name in square brackets, and then follow it with two colon characters (`::`), followed by the name of the method or static field.

In this recipe, you examine the assemblies loaded into PowerShell 7 and compare that with the behavior in Windows PowerShell. The recipe illustrates some of the differences between how PowerShell 7 and Windows PowerShell co-exist with .NET. You also look at a module and the assembly that implements the commands in the module.

Getting ready

You run this recipe on `SRV1`, a workgroup server on which you have installed PowerShell 7 and VS Code.

How to do it...

1. Counting loaded assemblies

```
$Assemblies = [System.AppDomain]::CurrentDomain.GetAssemblies()
"Assemblies loaded: {0:n0}" -f $Assemblies.Count
```

2. Viewing the first 10

```
$Assemblies | Select-Object -First 10
```

3. Checking assemblies in Windows PowerShell

```
$SB = {
  [System.AppDomain]::CurrentDomain.GetAssemblies()
}
$PS51 = New-PSSession -UseWindowsPowerShell
$AssIn51 = Invoke-Command -Session $PS51 -ScriptBlock $SB
"Assemblies loaded in Windows PowerShell: {0:n0}" -f $Assin51.Count
```

4. Viewing `Microsoft.PowerShell` assemblies

```
$AssIn51 |
  Where-Object FullName -Match "Microsoft\.Powershell" |
    Sort-Object -Property Location
```

5. Exploring the `Microsoft.PowerShell.Management` module

```
$Mod = Get-Module -Name Microsoft.PowerShell.Management -ListAvailable
$Mod  | Format-List
```

6. Viewing the module manifest

```
$Manifest = Get-Content -Path $Mod.Path
$Manifest | Select-Object -First 20
```

7. Discovering the module's assembly

```
Import-Module -Name Microsoft.PowerShell.Management
$Match = $Manifest | Select-String Modules
$Lube = $Match.Line
$DLL = ($Line -Split '"')[1]
Get-Item -Path $PSHOME\$DLL
```

8. Viewing associated loaded assembly

```
$Assemblies2 = [System.AppDomain]::CurrentDomain.GetAssemblies()
$Assemblies2 | Where-Object Location -match $DLL
```

9. Getting details of a PowerShell command inside a module DLL

```
$Commands = $Assemblies2
  Where-Object Location -match Commands.Management\.dll
$Commands.GetTypes() |
  Where-Object Name -match "Addcontentcommand$"
```

How it works...

In *step 1*, you use the `GetAssemblies()` method to return all the assemblies currently loaded by PowerShell. Then you output a count of the assemblies currently loaded, which looks like this:

```
PS C:\Foo> # 1. Counting loaded assemblies
PS C:\Foo> $Assemblies = [System.AppDomain]::CurrentDomain.GetAssemblies()
PS C:\Foo> "Assemblies loaded: {0:n0}" -f $Assemblies.Count
Assemblies loaded: 138
```

Figure 5.2: Counting the loaded assemblies

In *step 2*, you look at the first 10 assemblies returned, which looks like this:

```
PS C:\Foo> # 2. Viewing first 10
PS C:\Foo> $Assemblies | Select-Object -First 10

GAC    Version        Location
---    -------        --------
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Private.CoreLib.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\pwsh.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Runtime.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\Microsoft.PowerShell.ConsoleHost.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Management.Automation.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Threading.Thread.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Runtime.InteropServices.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Threading.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\Microsoft.Win32.Primitives.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Diagnostics.Process.dll
```

Figure 5.3: Viewing the first 10 assemblies

In *step 3*, you examine the assemblies loaded into Windows PowerShell 5.1, which looks like this:

```
PS C:\Foo> # 3. Checking assemblies in Windows PowerShell
PS C:\Foo> $SB = {
             [System.AppDomain]::CurrentDomain.GetAssemblies()
           }
PS C:\Foo> $PS51 = New-PSSession -UseWindowsPowerShell
PS C:\Foo> $Assin51 = Invoke-Command -Session $PS51 -ScriptBlock $SB
PS C:\Foo> "Assemblies loaded in Windows PowerShell: {0:n0}" -f $Assin51.Count
Assemblies loaded in Windows PowerShell: 16
```

Figure 5.4: Checking the assemblies in Windows PowerShell 5.1

With *step 4*, you examine the `Microsoft.PowerShell.*` assemblies in PowerShell 5.1, which looks like this:

```
PS C:\Foo> # 4. Viewing Microsoft.PowerShell assemblies
PS C:\Foo> $Assin51 |
             Where-Object FullName -Match "Microsoft\.Powershell" |
               Sort-Object -Property Location

GAC    Version        Location                                                                PSComputerName
---    -------        --------                                                                --------------
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.Consol… localhost
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.Securi… localhost
```

Figure 5.5: Viewing the Microsoft.PowerShell assemblies in Windows PowerShell

In *step 5*, you examine the details of the `Microsoft.PowerShell.Management` module, which contains numerous core commands in PowerShell. The output of this step looks like this:

```
PS C:\Foo> # 5. Exploring the Microsoft.PowerShell.Management module
PS C:\Foo> $Mod =
             Get-Module -Name Microsoft.PowerShell.Management -ListAvailable
PS C:\Foo> $Mod  | Format-List


Name              : Microsoft.PowerShell.Management
Path              : C:\program files\powershell\7\Modules\Microsoft.PowerShell.Manag
                    ement\Microsoft.PowerShell.Management.psd1
Description       :
ModuleType        : Manifest
Version           : 7.0.0.0
PreRelease        :
NestedModules     : {Microsoft.PowerShell.Commands.Management}
ExportedFunctions :
ExportedCmdlets   : {Add-Content, Clear-Content, Get-Clipboard, Set-Clipboard,
                    Clear-ItemProperty, Join-Path, Convert-Path, Copy-ItemProperty,
                    Get-ChildItem, Get-Content, Get-ItemProperty, Get-ItemPropertyValue,
                    Move-ItemProperty, Get-Location, Set-Location, Push-Location,
                    Pop-Location, New-PSDrive, Remove-PSDrive, Get-PSDrive, Get-Item,
                    New-Item, Set-Item, Remove-Item, Move-Item, Rename-Item, Copy-Item,
                    Clear-Item, Invoke-Item, Get-PSProvider, New-ItemProperty, Split-Path,
                    Test-Path, Test-Connection, Get-Process, Stop-Process, Wait-Process,
                    Debug-Process, Start-Process, Remove-ItemProperty, Rename-ItemProperty,
                    Resolve-Path, Get-Service, Stop-Service, Start-Service,
                    Suspend-Service, Resume-Service, Restart-Service, Set-Service,
                    New-Service, Remove-Service, Set-Content, Set-ItemProperty,
                    Restart-Computer, Stop-Computer, Rename-Computer, Get-ComputerInfo,
                    Get-TimeZone, Set-TimeZone, Get-HotFix, Clear-RecycleBin}
ExportedVariables :
ExportedAliases   : {gcb, gin, gtz, scb, stz}
```

Figure 5.6: Exploring the Microsoft.PowerShell.Management module in PowerShell 7.1

In *step 6*, you view the manifest for the `Microsoft.PowerShell.Management` module. The following screenshot shows the first 20 lines of the manifest:

```
PS C:\Foo> # 6. Viewing module manifest
PS C:\Foo> $MANIFEST = Get-Content -Path $MOD.Path
PS C:\Foo> $MANIFEST | Select-Object -First 20
@{
GUID="EEFCB906-B326-4E99-9F54-8B4BB6EF3C6D"
Author="PowerShell"
CompanyName="Microsoft Corporation"
Copyright="Copyright (c) Microsoft Corporation."
ModuleVersion="7.0.0.0"
CompatiblePSEditions = @("Core")
PowerShellVersion="3.0"
NestedModules="Microsoft.PowerShell.Commands.Management.dll"
HelpInfoURI = 'https://aka.ms/powershell71-help'
FunctionsToExport = @()
AliasesToExport = @("gcb", "gin", "gtz", "scb", "stz")
CmdletsToExport=@("Add-Content",
    "Clear-Content",
    "Get-Clipboard",
    "Set-Clipboard",
    "Clear-ItemProperty",
    "Join-Path",
    "Convert-Path",
    "Copy-ItemProperty",
```

Figure 5.7: Viewing the Microsoft.PowerShell.Management module manifest

In *step 7*, you extract the name of the DLL implementing the `Microsoft.PowerShell.Management` module and examine the location on disk, which looks like this:

```
PS C:\Foo> # 7. Discovering the module's assembly
PS C:\Foo> Import-Module -Name Microsoft.PowerShell.Management
PS C:\Foo> $Match = $Manifest | Select-String Modules
PS C:\Foo> $Line = $Match.Line
PS C:\Foo> $DLL = ($Line -Split '"')[1]
PS C:\Foo> Get-Item -Path $PSHOME\$DLL

    Directory: C:\Program Files\PowerShell\7

Mode          LastWriteTime    Length Name
----          -------------    ------ ----
-a---   06/11/2020    02:33   1119624 Microsoft.PowerShell.Commands.Management.dll
```

Figure 5.8: Discovering the Microsoft.PowerShell.Management module's assembly

In *step 8*, you find the assembly that contains the cmdlets in the `Microsoft.PowerShell.Management` module, which looks like this:

```
PS C:\Foo> # 8. Viewing associated loaded assembly
PS C:\Foo> $Assemblies2 = [appdomain]::CurrentDomain.GetAssemblies()
PS C:\Foo> $Assemblies2 | Where-Object Location -match $DLL

GAC   Version     Location
---   -------     --------
False v4.0.30319  C:\Program Files\PowerShell\7-preview\Microsoft.PowerShell.Commands.Management.dll
```

Figure 5.9: Viewing the associated loaded assembly

In *step 9*, you discover the name of the class that implements the `Add-Content` command, which looks like this:

```
PS C:\Foo> # 9. Getting details of a command
PS C:\Foo> $Commands  = $Assemblies2
               Where-Object Location -match Commands.Management\.dll
PS C:\Foo> $Commands.GetTypes() |
             Where-Object Name -match "Addcontentcommand$"

IsPublic IsSerial Name                BaseType
-------- -------- ----                --------
True     False    AddContentCommand   Microsoft.PowerShell.Commands.WriteContentCommandBase
```

Figure 5.10: Getting details of a command

There's more...

In this recipe, you have seen the .NET assemblies used by PowerShell. These consist of both the .NET Framework assemblies (that is, the BCLs) and the assemblies which implement PowerShell commands. For example, you find the `Add-Content` cmdlet in the `Microsoft.PowerShell.Management` module.

In *step 1*, you use the `GetAssemblies()` method to return all the assemblies currently loaded by PowerShell 7.1. As you can see, the syntax is different from calling PowerShell cmdlets.

In *steps 3* and *4*, you obtain and view the assemblies loaded by Windows PowerShell 5.1. As you can see, different assemblies are loaded by Windows PowerShell.

In *step 6*, you view the first 20 lines of the module manifest for the `Microsoft.PowerShell.Management` module. The output cuts off the complete list of cmdlets exported by the module and the long digital signature for the module manifest. In *Figure 5.7*, you can see that the command `Add-Content` is implemented within `Microsoft.PowerShell.Management.dll`.

In *step 7*, you discover the DLL, which implements inter alia the `Add-Content` cmdlet and in *step 8*, you can see that the assembly is loaded. In *step 9*, you discover that the `Add-Content` command is implemented by the `AddContentCommand` class within the assembly's DLL. For the curious, navigate to `https://github.com/PowerShell/PowerShell/blob/master/src/Microsoft.PowerShell.Commands.Management/commands/management/AddContentCommand.cs`, where you can read the source code for this cmdlet.

Examining .NET classes

With .NET, a class defines an object. Objects and object occurrences are fundamental to PowerShell, where cmdlets produce and consume objects. For example, the `Get-Process` command returns objects of the class `System.Diagnostics.Process`. If you use `Get-ChildItem` to return files and folders, the output is a set of objects based on the `System.IO.FileInfo` and `System.IO.DirectoryInfo` classes.

In most cases, your console activities and scripts make use of the objects created automatically by PowerShell commands. But you can also use the `New-Object` command to create occurrences of any class as necessary. This book shows numerous examples of creating an object using `New-Object`.

Within .NET, you have two kinds of object definitions: .NET **classes** and .NET **types**. A type defines a simple object that lives, at runtime, on your CPU's stack. Classes, being more complex, live in the global heap. The global heap is a large area of memory which .NET uses to hold object occurrences during a PowerShell session. In almost all cases, the difference between type and class is not overly relevant to IT professionals using PowerShell.

After a script or even a part of a script has run, .NET can tidy up the global heap in a process known as **garbage collection** (**GC**). The garbage collection process is also not important for IT professionals in most cases. The scripts you see in this book, for example, are not generally impacted by the garbage collection process, nor are most production scripts. For more information on the garbage collection process in .NET, see `https://docs.microsoft.com/dotnet/standard/garbage-collection/`.

There are cases where the GC process can impact on performance. For example, the `System.Array` class creates objects of fixed length. If you add an item to an array, .NET creates a new copy of the array (including the addition) and removes the old one. If you are adding a few items to the array, the performance hit is negligible. But if you are adding millions of occurrences, the performance can suffer significantly. To avoid this, you can just use the `ArrayList` class, which supports adding/removing items from an array without the performance penalty. For more information on GC and performance, see `https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/performance`.

.NET 5.0 features many improvements to the garbage collection process. You can read more about the GC improvements in .NET 5.0 here: `https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/`.

In .NET, occurrences of every class or type can include members, including properties, methods, and events. A **property** is an attribute of an occurrence of a class. An occurrence of the `System.IO.FileInfo` object, for example, has a `FullName` property. A **method** is effectively a function you can call which can do something to an object occurrence. You look at .NET methods in more details in the *Leveraging .NET methods* recipe. An **event** is something that can happen to an object occurrence, such when an event is generated when a Windows process has completed. .NET events are not covered in this book, although using WMI events is described in *Chapter 15*, *Managing with Windows Management Instrumentation*, in the *Managing WMI events* recipe.

You can quickly determine an object's class (or type) by piping the output of any cmdlet, or an object, to the `Get-Member` cmdlet. The `Get-Member` cmdlet uses a feature of .NET, **reflection**, to look inside and give you a definitive statement of what that object contains. This feature is invaluable – instead of guessing where, in some piece of string output, your script can find the full name of a file, you can discover the `FullName` property, a string, or the `Length` property, which is unambiguously an integer. Reflection and the `Get-Member` cmdlet help you to discover the properties and other members of a given object.

.NET classes can have **static properties** and **static methods**. Static properties/methods are aspects of the class of a whole as opposed to a specific class instance. A static property is a fixed constant value, such as the maximum and minimum values for a 32-bit signed integer or the value of pi. A static method is one that is independent of any specific instance. For example, the `Parse()` method of the INT32 class can parse a string to ensure it is a value 32-bit signed integer. In most cases, you use static methods to create object instances or to do something related to the class.

In this recipe, you look at some everyday objects created automatically by PowerShell. You also examine the static fields of the `[Int]` .NET class.

Getting ready

You run this recipe on `SRV1`, a workgroup host running Windows Server Datacenter edition. This host has PowerShell and VS Code installed.

How to do it...

1. Creating a `FileInfo` object

```
$FILE = Get-ChildItem -Path $PSHOME\pwsh.exe
$FILE
```

2. Discovering the underlying class

```
$TYPE = $FILE.GetType().FullName
".NET Class name: $TYPE"
```

3. Getting member types of the `FileInfo` object

```
$File |
  Get-Member |
    Group-Object -Property MemberType |
      Sort-Object -Property Count -Descending
```

4. Discovering properties of a Windows service

```
Get-Service |
  Get-Member -MemberType Property
```

5. Discovering the underlying type of an integer

```
$I = 42
$IntType  = $I.GetType()
$TypeName = $IntType.FullName
$BaseType = $IntType.BaseType.Name
".Net Class name      : $TypeName"
".NET Class Base Type : $BaseType"
```

6. Looking at process objects

```
$PWSH = Get-Process -Name pwsh |
  Select-Object -First 1
$PWSH |
  Get-Member |
    Group-Object -Property MemberType |
      Sort-Object -Property Count -Descending
```

7. Looking at static properties of a class

```
$Max = [Int32]::MaxValue
$Min = [Int32]::MinValue
"Minimum value [$Min]"
"Maximum value [$Max]"
```

How it works...

In *step 1*, you use the `Get-ChildItem` cmdlet to return an object representing the PowerShell 7 executable file, with output like this:

```
PS C:\Foo> # 1. Creating a FileInfo object
PS C:\Foo> $FILE = Get-ChildItem -Path $PSHOME\pwsh.exe
PS C:\Foo> $FILE

    Directory: C:\Program Files\PowerShell\7

Mode         LastWriteTime  Length Name
----         -------------  ------ ----
-a---  06/11/2020    02:33  280456 pwsh.exe
```

Figure 5.11: Creating a FileInfo object

You can determine the class name of an object by using the `GetType()` method. This method is present on every object and returns information about the object's type.

In *step 2*, you discover and display a full class name, which looks like this:

```
PS C:\Foo> # 2. Discovering the underlying class
PS C:\Foo> $TYPE = $FILE.GetType().FullName
PS C:\Foo> ".NET Class: $TYPE"
.NET Class: System.IO.FileInfo
```

Figure 5.12: Discovering the underlying class

In *step 3*, you use `Get-Member` to display the different member types contained within a `FileInfo` object, with output like this:

```
PS C:\Foo> # 3. Getting member types of FileInfo object
PS C:\Foo> $File |
             Get-Member |
               Group-Object -Property MemberType |
                 Sort-Object -Property Count -Descending

Count Name             Group
----- ----             -----
   21 Method           {System.IO.StreamWriter AppendText(), System.IO.FileInfo…
   15 Property         {System.IO.FileAttributes Attributes {get;set;}, datetim…
    6 NoteProperty     {string PSChildName=pwsh.exe, PSDriveInfo PSDrive=C, boo…
    4 CodeProperty     {System.String LinkType{get=GetLinkType;}, System.String…
    2 ScriptProperty   {System.Object BaseName {get=if ($this.Extension.Length …
```

Figure 5.13: Getting member types of the FileInfo object

In *step 4*, you examine objects returned from the `Get-Service` cmdlet. The output of this step looks like this:

```
PS C:\Foo> # 4. Discovering properties of a Windows service
PS C:\Foo> Get-Service |
             Get-Member -MemberType Property

   TypeName: System.Service.ServiceController#StartupType

Name                  MemberType Definition
----                  ---------- ----------
BinaryPathName        Property   System.String {get;set;}
CanPauseAndContinue   Property   bool CanPauseAndContinue {get;}
CanShutdown           Property   bool CanShutdown {get;}
CanStop               Property   bool CanStop {get;}
Container             Property   System.ComponentModel.IContainer Container {get;}
DelayedAutoStart      Property   System.Boolean {get;set;}
DependentServices     Property   System.ServiceProcess.ServiceController[] DependentServices {get;}
Description           Property   System.String {get;set;}
DisplayName           Property   string DisplayName {get;set;}
MachineName           Property   string MachineName {get;set;}
ServiceHandle         Property   System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName           Property   string ServiceName {get;set;}
ServicesDependedOn    Property   System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType           Property   System.ServiceProcess.ServiceType ServiceType {get;}
Site                  Property   System.ComponentModel.ISite Site {get;set;}
StartType             Property   System.ServiceProcess.ServiceStartMode StartType {get;}
StartupType           Property   Microsoft.PowerShell.Commands.ServiceStartupType {get;set;}
Status                Property   System.ServiceProcess.ServiceControllerStatus Status {get;}
UserName              Property   System.String {get;set;}
```

Figure 5.14: Discovering properties of a Windows service

In *step 5*, you examine the details of an integer, a `System.Int32` data type. You use the `GetType()` method, which is present on all .NET objects, to return the variable's type information, including the full class name of the class. The output looks like this:

```
PS C:\Foo> # 5. Discovering the underlying type of an integer
PS C:\Foo> $I = 42
PS C:\Foo> $IntType  = $I.GetType()
PS C:\Foo> $TypeName = $IntType.Name
PS C:\Foo> $BaseType = $IntType.BaseType.Name
PS C:\Foo> ".Net Class name      : $TypeName"
PS C:\Foo> ".NET Class Base Type : $BaseType"
.NET Class name      : System.Int32
.NET Class base type : ValueType
```

Figure 5.15: Examining type information of an integer

In *step 6*, you examine the member types of a `System.Diagnostics.Process` object. The `Get-Process` cmdlet returns objects of this class. The output of this step looks like this:

```
PS C:\Foo> # 6. Looking at process objects
PS C:\Foo> $PWSH = Get-Process -Name pwsh |
             Select-Object -First 1
PS C:\Foo> $PWSH |
             Get-Member |
               Group-Object -Property MemberType |
                 Sort-Object -Property Count -Descending

  Count Name             Group
----- ----             -----
   52 Property         {int BasePriority {get;}, System.ComponentModel.IContainer Container {ge…
   19 Method           {void BeginErrorReadLine(), void BeginOutputReadLine(), void CancelError…
    8 ScriptProperty   {System.Object CommandLine {get=…
    7 AliasProperty    {Handles = Handlecount, Name = ProcessName, NPM = NonpagedSystemMemorySi…
    4 Event            {System.EventHandler Disposed(System.Object, System.EventArgs), System.D…
    2 PropertySet      {PSConfiguration {Name, Id, PriorityClass, FileVersion}, PSResources {Na…
    1 CodeProperty     {System.Object Parent{get=GetParentProcess;}}
    1 NoteProperty     {string __NounName=Process}
```

Figure 5.16: Looking at member types of a process object

In *step 7*, you examine two static values of the `System.Int32` class. These two fields hold the largest and smallest values, respectively, that a 32-bit signed integer can hold. The output looks like this:

```
PS C:\Foo> # 7. Looking at static properties within a class
PS C:\Foo> $Max = [Int32]::MaxValue
PS C:\Foo> $Min = [Int32]::MinValue
PS C:\Foo> "Minimum value [$Min]"
PS C:\Foo> "Maximum value [$Max]"
Minimum value [-2147483648]
Maximum value [2147483647]
```

Figure 5.17: Examining static values of the System.Int32 class

There's more...

In *step 1*, you create an object representing `pwsh.exe`. This object's full type name is `System.IO.FileInfo`. In .NET, classes live in namespaces and, in general, namespaces equate to DLLs. In this case, the object is in the `System.IO` namespace, and the class is contained in `System.IO.FileSystem.DLL`. You can discover namespace and DLL details by examining the class's documentation – in this case, `https://docs.microsoft.com/dotnet/api/system.io.fileinfo?view=net-5.0`.

As you use your favourite search engine to learn more about .NET classes that might be useful, note that many sites describe the class without the namespace, simply as the `FileInfo` class, while others spell out the class as `System.IO.FileInfo`. If you are going to be using .NET class names in your scripts, a best practice is to spell out the full class name.

Leveraging .NET methods

With .NET, a method is some action that a .NET object occurrence, or the class, can perform. These methods form the basis for many PowerShell cmdlets. For example, you can stop a Windows process by using the `Stop-Process` cmdlet. The cmdlet then uses the `Kill()` method of the associated process object. As a general best practice, you should use cmdlets wherever possible. You should only use .NET classes and methods directly where there is no alternative.

.NET methods can be beneficial for performing operations which have no PowerShell cmdlets. And it can be useful too from the command line; for example, when you wish to kill a process. IT professionals are all too familiar with processes that are not responding and need to be killed, something you can do at the GUI using Task Manager. Or with PowerShell, you can use the `Stop-Process` cmdlet, as discussed above. At the command line, where brevity is useful, you can use `Get-Process` to find the process you want to stop and pipe the output to each process's `Kill()` method. PowerShell then calls the object's `Kill()` method. Of course, to help IT professionals, the PowerShell team created the `Kill` alias to the `Stop-Process` cmdlet. In practice, it's four characters to type versus 12 (or eight if you are using tab completion to its best effect). At the command line, piping to the `kill` method (where you don't even have to use the open/close parentheses!) is just faster and risks fewer typos. It is a good shortcut at the command line – but avoid it in production code.

Another great example is encrypting files. Windows supports the NTFS **Encrypting File System** (**EFS**) feature. EFS enables you to encrypt or decrypt files on your computer with the encryption based on X.509 certificates. For details on the EFS and how it works, see `https://docs.microsoft.com/windows/win32/fileio/file-encryption`.

At the time of writing, there are no cmdlets to encrypt or decrypt files. The `System.IO.FileInfo` class, however, has two methods you can use: `Encrypt()` and `Decrypt()`. These methods encrypt and decrypt a file based on EFS certificates. You can use these .NET methods to encrypt or decrypt a file without having to use the GUI.

As you saw in the *Examining .NET classes* recipe, you can pipe any object to the `Get-Member` cmdlet to discover the methods for the object. Discovering the specific property names and property value types is simple and easy – no guessing or prayer-based text parsing, so beloved by Linux admins.

Getting ready

You run this recipe on SRV1 after loading PowerShell 7.1 and VS Code.

How to do it...

1. Starting Notepad

```
notepad.exe
```

2. Obtaining methods on the Notepad process

```
$Notepad = Get-Process -Name Notepad
$Notepad | Get-Member -MemberType method
```

3. Using the Kill() method

```
$Notepad |
  ForEach-Object {$_.Kill()}
```

4. Confirming the Notepad process is destroyed

```
Get-Process -Name Notepad
```

5. Creating a new folder and some files

```
$Path = 'C:\Foo\Secure'
New-Item -Path $Path -ItemType directory -ErrorAction SilentlyContinue  |
  Out-Null
1..3 | ForEach-Object {
  "Secure File" | Out-File "$Path\SecureFile$_.txt"
}
```

6. Viewing files in the $Path folder

```
$Files = Get-ChildItem -Path $Path
$Files | Format-Table -Property Name, Attributes
```

7. Encrypting the files

```
$Files| ForEach-Object Encrypt
```

8. Viewing file attributes

```
Get-ChildItem -Path $Path |
  Format-Table -Property Name, Attributes
```

9. Decrypting and viewing the files

```
$Files| ForEach-Object {
  $_.Decrypt()
}
Get-ChildItem -Path $Path |
  Format-Table -Property Name, Attributes
```

How it works...

In *step 1*, which produces no output, you start `Notepad.exe`. This creates a process which you can examine and use.

In *step 2*, you obtain Notepad's process object and examine the methods available to you. The output looks like this:

```
PS C:\Foo> # 2. Obtaining methods on the Notepad process
PS C:\Foo> $Notepad = Get-Process -Name Notepad
PS C:\Foo> $Notepad | Get-Member -MemberType Method

   TypeName: System.Diagnostics.Process

Name                      MemberType Definition
----                      ---------- ----------
BeginErrorReadLine        Method     void BeginErrorReadLine()
BeginOutputReadLine       Method     void BeginOutputReadLine()
CancelErrorRead           Method     void CancelErrorRead()
CancelOutputRead          Method     void CancelOutputRead()
Close                     Method     void Close()
CloseMainWindow           Method     bool CloseMainWindow()
Dispose                   Method     void Dispose(), void IDisposable.Dispose()
Equals                    Method     bool Equals(System.Object obj)
GetHashCode               Method     int GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetType                   Method     type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
Kill                      Method     void Kill(), void Kill(bool entireProcessTree)
Refresh                   Method     void Refresh()
Start                     Method     bool Start()
ToString                  Method     string ToString()
WaitForExit               Method     void WaitForExit(), bool WaitForExit(int milliseconds)
WaitForExitAsync          Method     System.Threading.Tasks.Task WaitForExitAsync(System.Threading.Cancellati…
WaitForInputIdle          Method     bool WaitForInputIdle(), bool WaitForInputIdle(int milliseconds)
```

Figure 5.18: Examining methods on the Notepad process

In *step 3*, you use the `Kill()` method in the `System.Diagnostics.Process` object to stop the Notepad process. This step produces no output. In *step 4*, you confirm that you have stopped the Notepad process, with output like this:

```
PS C:\Foo> # 4. Confirming Notepad process is destroyed
PS C:\Foo> Get-Process -Name Notepad
Get-Process:
Line |
   2 |  Get-Process -Name Notepad
     |  ~~~~~~~~~~~~~~~~~~~~~~~~~
     | Cannot find a process with the name "Notepad". Verify the process name
       and call the cmdlet again.
```

Figure 5.19: Confirming the Notepad process has been destroyed

To illustrate other uses of .NET methods, you create a folder and three files within the folder in *step 5*. Creating these files generates no output. In *step 6*, you use the `Get-ChildItem` cmdlet to retrieve details about these three files, including all file attributes, which looks like this:

```
PS C:\Foo> # 6. Viewing files in $Path folder
PS C:\Foo> $Files = Get-ChildItem -Path $Path
PS C:\Foo> $Files | Format-Table -Property Name, Attributes

Name            Attributes
----            ----------
SecureFile1.txt    Archive
SecureFile2.txt    Archive
SecureFile3.txt    Archive
```

Figure 5.20: Viewing the files in the $Path folder

In *step 7*, you use the encrypt method to encrypt the files, generating no output at the command line. In *step 8*, you view again the attributes of the files, which looks like this:

```
PS C:\Foo> # 8. Viewing file attributes
PS C:\Foo> Get-ChildItem -Path $Path |
             Format-Table -Property Name, Attributes

Name                    Attributes
----                    ----------
SecureFile1.txt Archive, Encrypted
SecureFile2.txt Archive, Encrypted
SecureFile3.txt Archive, Encrypted
```

Figure 5.21: Viewing the file attributes

Finally, with *step 9*, you decrypt the files using the `Decrypt()` method. You then use the `Get-ChildItem` cmdlet to view the file attributes, which allows you to determine that the files are not encrypted. The output of this step looks like this:

```
PS C:\Foo> # 9. Decrypting and viewing the files
PS C:\Foo> $Files| ForEach-Object {
             $_.Decrypt()
           }
PS C:\Foo> Get-ChildItem -Path $Path |
             Format-Table -Property Name, Attributes

Name            Attributes
----            ----------
SecureFile1.txt    Archive
SecureFile2.txt    Archive
SecureFile3.txt    Archive
```

Figure 5.22: Decrypting and viewing the file attributes again

There's more...

In *step 2*, you view the methods you can invoke on a process object. One of those methods is the `Kill()` method, as you can see in Figure 5.18. In *step 3*, you use that method to stop the Notepad process. The `Kill()` method is an instance method, meaning you invoke it to kill (stop) a specific process. You can read more about this .NET method at `https://docs.microsoft.com/dotnet/api/system.diagnostics.process.kill`.

The output in *step 4* illustrates an error occurring within VS Code. If you use the PowerShell 7 console, you may see slightly different output, although with the same actual error message.

In *steps 7* and *9*, you use the `FileInfo` objects (created by `Get-ChildItem`) and call the `Encrypt()` and `Decrypt()` methods. These steps demonstrate using .NET methods to achieve some objective in the absence of specific cmdlets. Best practice suggests always using cmdlets where you can. You should also note that the syntax in the two steps is different. In *step 7*, you use a more modern syntax which calls the `Encrypt` method for each file. In *step 9*, you use an older syntax that does the same thing, albeit with more characters. Both syntax methods work.

In *step 7*, while you get no console output, Windows may generate a popup (aka Toast) to tell you that you should back up your encryption key. This is a good idea because if you lose the key, you could lose all access to the file.

Creating a C# extension

For most day-to-day operations, the commands provided by PowerShell from Windows features, or third-party modules, give you all the functionality you need. In some cases, as you saw in the *Leveraging .NET methods* recipe, commands do not exist to achieve your goal. In those cases, you can make use of the methods provided by .NET.

There are also cases where you need to perform more complex operations without PowerShell cmdlet or direct .NET support. You may, for example, have a component of an ASP.NET web application, written in C# but which you now wish to repurpose for administrative scripting purposes.

PowerShell makes it easy to add a class, based on .NET language source code, into a PowerShell session. You supply the C# code, and PowerShell creates a .NET class that you can use in the same way you use .NET methods (and using virtually the same syntax). To do this, you use the `Add-Type` cmdlet and specify the C# code for your class/type(s). PowerShell compiles the code and loads the resultant class into your PowerShell session.

An essential aspect of .NET methods is that a method can have multiple definitions or calling sequences. Known as **method overloads**, these multiple definitions allow you to invoke a method using different sets of parameters. This is not dissimilar to PowerShell's use of parameter sets. For example, the `System.String` class, which PowerShell uses to hold strings, contains the `Trim()` method. You use that method to remove extra characters, usually space characters, from the start or end of a string (or both). The `Trim()` method has three different definitions, which you view in this recipe. Each overloaded definition trims characters from a string slightly differently. To view more details on this method, and the three overloaded definitions, see `https://docs.microsoft.com/dotnet/api/system.string.trim?view=net-5.0/`.

In this recipe, you create and use two simple classes, each with static methods.

Getting ready

You run this recipe on `SRV1`, a workgroup system running Windows Server Datacenter edition. You must have loaded PowerShell 7 and VS Code onto this host.

How to do it...

1. Examining the overloaded method definition

```
("a string").Trim
```

2. Creating a C# class definition in a here-string

```
$NewClass = @"
namespace Reskit {
   public class Hello {
     public static void World() {
         System.Console.WriteLine("Hello World!");
     }
   }
  }
"@
```

3. Adding the type into the current PowerShell session

```
Add-Type -TypeDefinition $NewClass
```

4. Examining the method definition

```
[Reskit.Hello]::World
```

5. Using the class's method

```
[Reskit.Hello]::World()
```

6. Extending the code with parameters

```
$NewClass2 = @"
using System;
using System.IO;
namespace Reskit {
  public class Hello2  {
    public static void World() {
      Console.WriteLine("Hello World!");
    }
    public static void World(string name) {
      Console.WriteLine("Hello " + name + "!");
    }
  }
}
"@
```

7. Adding the type into the current PowerShell session

```
Add-Type -TypeDefinition $NewClass2 -Verbose
```

8. Viewing method definitions

```
[Reskit.Hello2]::World
```

9. Calling new method with no parameters specified

```
[Reskit.Hello2]::World()
```

10. Calling new method with a parameter

```
[Reskit.Hello2]::World('Jerry')
```

How it works...

In *step 1*, you examine the different method definitions for the `Trim()` method. The output, showing the different overloaded definitions, looks like this:

```
PS C:\Foo> # 1. Examining overloaded method definition
PS C:\Foo> ("a string").Trim

OverloadDefinitions
-------------------
string Trim()
string Trim(char trimChar)
string Trim(Params char[] trimChars)
```

Figure 5.23: Examining different method definitions for Trim()

In *step 2*, you create a class definition and store the definition in a variable. In *step 3*, you add this class definition into your current PowerShell workspace. These two steps produce no output.

In *step 4*, you use the new method to observe the definitions available to you, which looks like this:

```
PS C:\Foo> # 4. Examining method definition
PS C:\Foo> [Reskit.Hello]::World

OverloadDefinitions
-------------------
static void World()
```

Figure 5.24: Examining method definitions

In *step 5*, you use the `World()` method, whose output is as follows:

```
PS C:\Foo> # 5. Using the class's method
PS C:\Foo> [Reskit.Hello]::World()
Hello World!
```

Figure 5.25: Using the World() method

In *steps 6* and *7*, you create another new class definition, this time with two overloaded definitions to the `World()` method. These two steps produce no output. In *step 8*, you view the definitions for the `World()` method within the `Reskit.Hello2` class, with output like this:

```
PS C:\Foo> # 8. Viewing method definitions
PS C:\Foo> [Reskit.Hello2]::World

OverloadDefinitions
-------------------
static void World()
static void World(string name)
```

Figure 5.26: Viewing the World() method definitions

In *step 9*, you invoke the `World()` method, specifying no parameters, which produces this output:

```
PS C:\Foo> # 9. Calling with no parameters specified
PS C:\Foo> [Reskit.Hello2]::World()
Hello World!
```

Figure 5.27: Calling World() with no parameters specified

In *step 10*, you invoke the `World()` method, specifying a single string parameter, which produces this output:

```
PS C:\Foo> # 10. Calling new method with a parameter
PS C:\Foo> [Reskit.Hello2]::World('Jerry')
Hello Jerry!
```

Figure 5.28: Calling World() while specifying a single string parameter

There's more...

In *step 1*, you examine the different definitions of a method – in this case, the `Trim()` method of the `System.String` class. There are several ways you discover method overloads. In this step, you create a pseudo-object containing a string and then look at the method definitions. PowerShell creates an unnamed object (in the .NET managed heap) which it immediately destroys after the statement completes. At some point later in time, .NET runs the GC process to reclaim the memory used by this temporary object and reorganizes the managed heap.

As you can see, there are three overloaded definitions – three different ways you can invoke the `Trim()` method. You use the first overloaded definition to remove both leading and trailing space characters from a string, probably the most common usage of `Trim()`. With the second definition, you specify a specific character and .NET removes any leading or trailing occurrences of that character. With the third definition, you specify an array of characters to trim from the start or end of the string. In most cases, the last two definitions are less useful, although it depends on your use cases. The extra flexibility is useful.

In *step 3*, you use the `Add-Type` cmdlet to add the class definition into your current workspace. The cmdlet compiles the C# code, creates, and then loads a DLL, which then enables you to use the classes. If you intend to use this add-in regularly, then you could add *steps 2* and *3* into your PowerShell profile file or a production script. Alternatively, if you use this method within a script, you could add the steps into the script.

As you can see from the C# code in *step 6*, you can add multiple definitions to a method. With the classes within .NET and its BCLs, or in your own code, overloaded methods can provide a great deal of value. The .NET method documentation, which you can find at `https://docs.microsoft.com`, is mainly oriented toward developers rather than IT professionals. If you have issues with a .NET method, feel free to fire off a query on one of the many PowerShell support forums, such as the PowerShell group at Spiceworks: `https://community.spiceworks.com/programming/powershell`.

Creating a PowerShell cmdlet

As noted previously, for most operations, the commands and cmdlets available to you natively provide all the functionality you need (in most cases). In the *Creating a C# extension* recipe, you saw how you could create a class definition and add it into PowerShell. In some cases, you may wish to expand on the class definition and create your own cmdlet.

Creating a compiled cmdlet requires you to either use a tool such as Visual Studio or use the free tools provided by Microsoft as part of the .NET Core **Software Development Kit** (**SDK**). Visual Studio, whether the free community edition or the commercial releases, is a rich and complex tool whose inner workings are well outside the scope of this book. The free tools in the SDK are more than adequate to help you create a cmdlet using C#.

As in the *Creating a C# extension* recipe, an important question to ask is: why and when should you create a cmdlet? Aside from the perennial "because you can" excuse, there are reasons why an extension or a cmdlet might be a good idea. You can create a cmdlet to improve performance – for some operations, a compiled cmdlet is just faster than doing operations using a PowerShell script. For some applications, using a cmdlet is to perform actions that are difficult or not possible directly in PowerShell, including asynchronous operations and the use of a **Language Independent Query** (**LINQ**). A developer could write the cmdlet in C#, allowing you to use it easily with PowerShell.

In order to create a cmdlet, you need to install the Windows SDK. The SDK contains all the tools you need to create a cmdlet. The SDK is a free download you get via the internet.

In this recipe, you load the Windows SDK and use it to create a new PowerShell cmdlet. The installation of the .NET SDK requires you to navigate to a web page, download, and then run the SDK installer.

Getting ready

You run this recipe on `SRV1`, a workgroup system running Windows Server Datacenter edition. You must have loaded PowerShell 7 and VS Code onto this host.

How to do it...

1. Installing the .NET 5.0 SDK

   Open your browser, navigate to `https://dotnet.microsoft.com/download/`, and follow the GUI to download and run the SDK installer. You can see this process in the *How it works...* section below.

2. Creating the `cmdlet` folder

```
New-Item -Path C:\Foo\Cmdlet -ItemType Directory -Force
Set-Location C:\Foo\Cmdlet
```

3. Creating a new class library project

```
dotnet new classlib --name SendGreeting
```

4. Viewing contents of the new folder

```
Set-Location -Path .\SendGreeting
Get-ChildItem
```

5. Creating and displaying the `global.json` file

```
dotnet new globaljson
Get-Content .\global.json
```

6. Adding the PowerShell package

```
dotnet add package PowerShellStandard.Library
```

7. Creating the cmdlet source file

```
$Cmdlet = @"
using System.Management.Automation;  // Windows PowerShell assembly.
namespace Reskit
{
  // Declare the class as a cmdlet
  // Specify verb and noun = Send-Greeting
  [Cmdlet(VerbsCommunications.Send, "Greeting")]
  public class SendGreetingCommand : PSCmdlet
  {
    // Declare the parameters for the cmdlet.
    [Parameter(Mandatory=true)]
    public string Name
    {
      get { return name; }
      set { name = value; }
    }
    private string name;
    // Override the ProcessRecord method to process the
    // supplied name and write a greeting to the user by
    // calling the WriteObject method.
    protected override void ProcessRecord()
    {
      WriteObject("Hello " + name + " - have a nice day!");
    }
  }
}
```

```
"@
$Cmdlet | Out-File .\SendGreetingCommand.cs
```

8. Removing the unused class file

```
Remove-Item -Path .\Class1.cs
```

9. Building the cmdlet

```
dotnet build
```

10. Importing the DLL holding the cmdlet

```
$DLLPath = '.\bin\Debug\net5.0\SendGreeting.dll'
Import-Module -Name $DLLPath
```

11. Examining the module's details

```
Get-Module SendGreeting
```

12. Using the cmdlet:

```
Send-Greeting -Name "Jerry Garcia"
```

How it works...

In *step 1*, you open your browser and navigate to the .NET Download page, which looks like this:

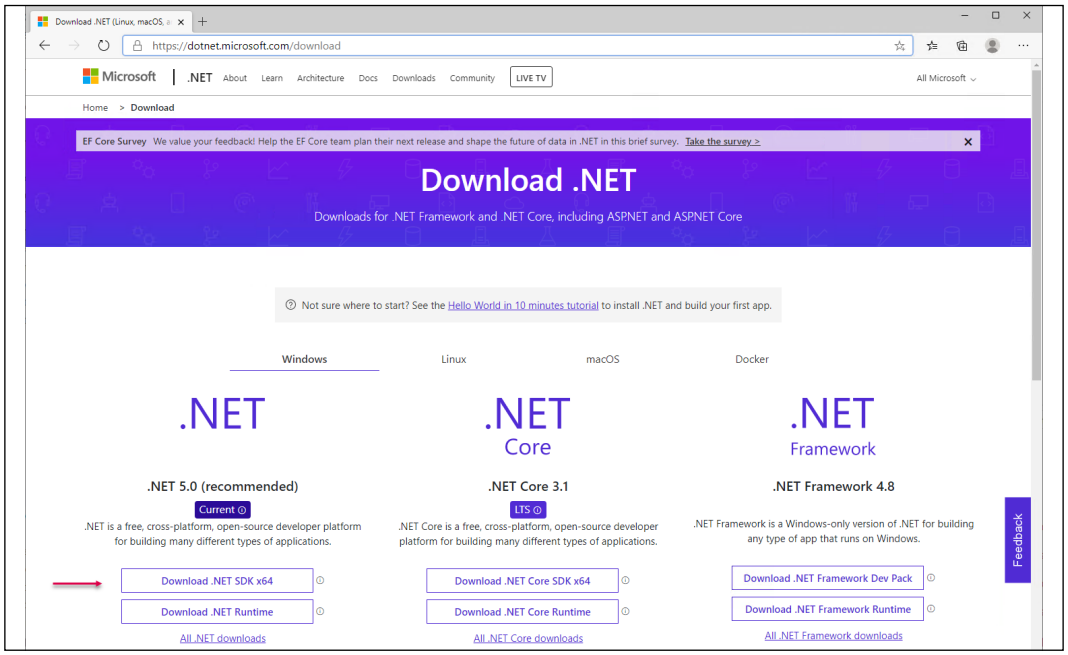

Figure 5.29: The .NET Download page

Click, as shown, on the link to **Download the .NET SDK X64** to download the .NET SDK installer program. Ensure you download the .NET 5.0 SDK, which looks like this:

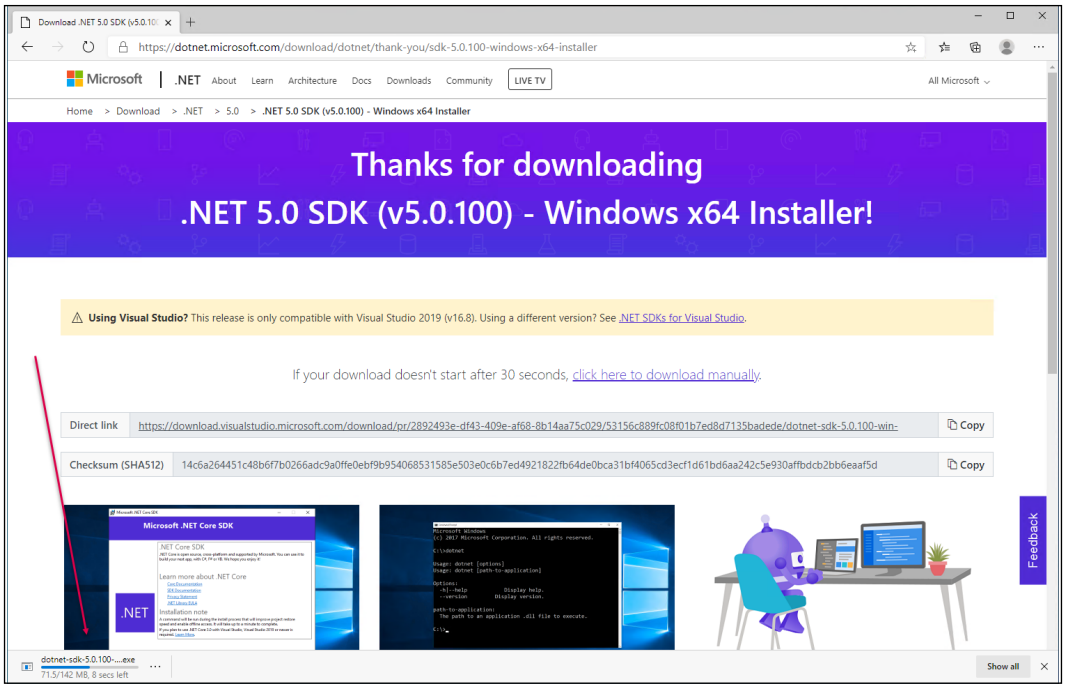

Figure 5.30: Downloading the .NET SDK installer

The installer program is around 150 MB, so it may take a while to download, depending on your internet connection speed. When the download is complete, you see the download indication look like this:

Figure 5.31: Installer download complete

If you click on the **Open file** link, you launch the installer and see the installer's opening page:

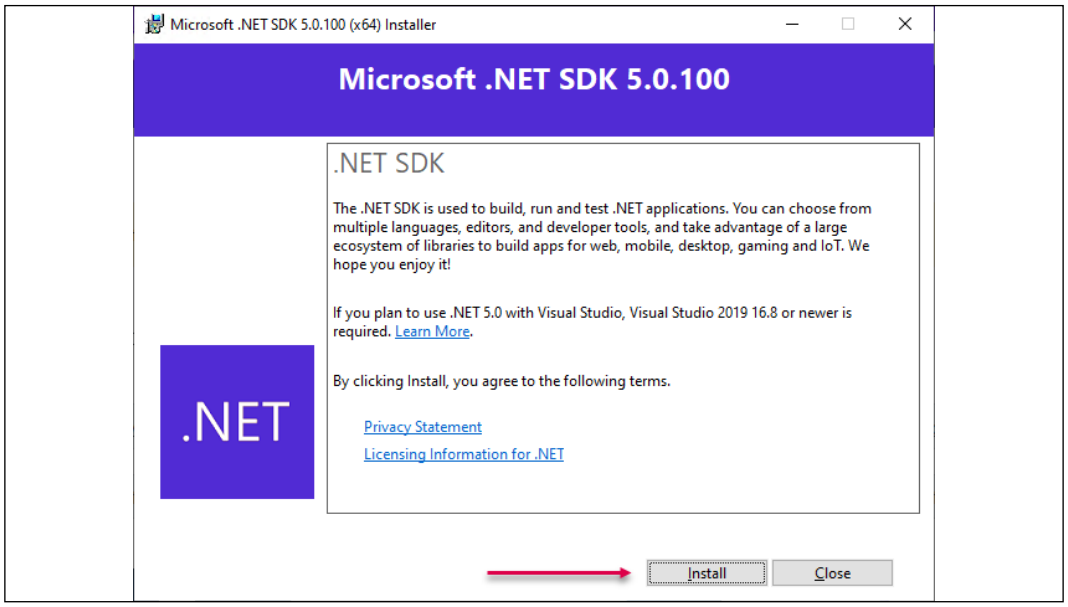

Figure 5.32: The .NET SDK installer

When you click on the **Install** button, as shown, the installer program completes the setup of the .NET SDK. When setup is complete, you see the final screen:

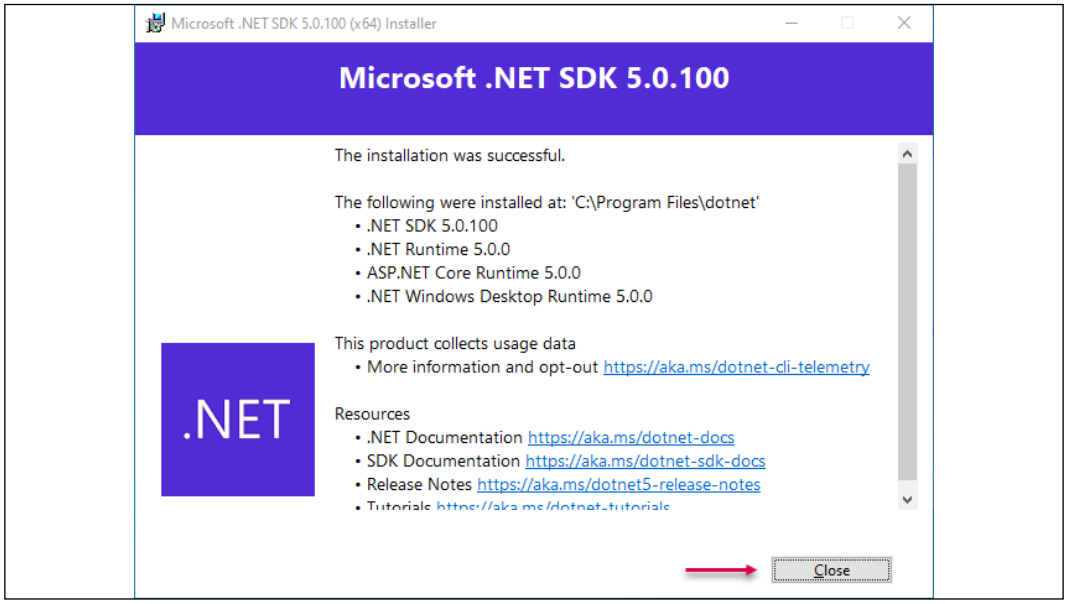

Figure 5.33: Installation successful screen

Click on **Close** to complete the setup. If you had a PowerShell console (or VS Code) open when you did the SDK installation, close then reopen it. The installer updates your system's `Path` variable so you can find the tools, which requires a restart of PowerShell to take effect.

In *step 2*, you create a new folder to hold your new cmdlet and use `Set-Location` to move into that folder. The output from this step looks like this:

```
PS C:\Foo> # 2. Creating the cmdlet folder
PS C:\Foo> New-Item -Path C:\Foo\Cmdlet -ItemType Directory -Force

    Directory: C:\Foo

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          01/12/2020    20:34                Cmdlet

PS C:\Foo> Set-Location C:\Foo\Cmdlet
PS C:\Foo\Cmdlet>
```

Figure 5.34: Creating the cmdlet folder and moving into it

In *step 3*, you create a new class library project using the `dotnet.exe` command, which looks like this:

```
PS C:\Foo\Cmdlet> # 3. Creating a new class library project
PS C:\Foo\Cmdlet> dotnet new classlib --name SendGreeting
The template "Class library" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on SendGreeting\SendGreeting.csproj...
  Determining projects to restore...
  Restored C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj (in 92 ms).
Restore succeeded.
```

Figure 5.35: Creating a new class library project

In *step 4*, you view the contents of the folder created by the previous step. The output looks like this:

```
PS C:\Foo\Cmdlet> # 4. Viewing contents of new folder
PS C:\Foo\Cmdlet> Set-Location -Path .\SendGreeting
PS C:\Foo\Cmdlet\SendGreeting> Get-ChildItem

    Directory: C:\Foo\Cmdlet\SendGreeting

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          01/12/2020    20:40                obj
-a---          01/12/2020    20:40             89 Class1.cs
-a---          01/12/2020    20:40            137 SendGreeting.csproj
```

Figure 5.36: Viewing the contents of the new folder

In *step 5*, you create and then view a new `global.json` file. This JSON file, among other things, specifies which version of .NET is to be used to build your new cmdlet. The output of this step looks like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 5. Creating and displaying global.json
PS C:\Foo\Cmdlet\SendGreeting> dotnet new globaljson
The template "global.json file" was created successfully.
PS C:\Foo\Cmdlet\SendGreeting> Get-Content -Path .\global.json

{
  "sdk": {
    "version": "5.0.100"
  }
}
```

Figure 5.37: Creating and displaying a new global.json file

In *step 6*, you add the PowerShell package to your project. This step enables you to use the classes in the package to build your cmdlet. The output looks like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 6. Adding PowerShell package
PS C:\Foo\Cmdlet\SendGreeting> dotnet add package PowerShellStandard.Library
  Determining projects to restore...
  Writing C:\Users\Administrator\AppData\Local\Temp\1\tmpDF50.tmp
info : Adding PackageReference for package 'PowerShellStandard.Library' into project 'C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj'.
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/powershellstandard.library/index.json
info :   OK https://api.nuget.org/v3/registration5-gz-semver2/powershellstandard.library/index.json 413ms
info : Restoring packages for C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj...
info :   GET https://api.nuget.org/v3-flatcontainer/powershellstandard.library/index.json
info :   OK https://api.nuget.org/v3-flatcontainer/powershellstandard.library/index.json 400ms
info :   GET https://api.nuget.org/v3-flatcontainer/powershellstandard.library/5.1.0/powershellstandard.library.5.1.0.nupkg
info :   OK https://api.nuget.org/v3-flatcontainer/powershellstandard.library/5.1.0/powershellstandard.library.5.1.0.nupkg 14ms
info : Installing PowerShellStandard.Library 5.1.0.
info : Package 'PowerShellStandard.Library' is compatible with all the specified frameworks in project 'C:\Foo\Cmdlet\SendGreeting\SendGr
eeting.csproj'.
info : PackageReference for package 'PowerShellStandard.Library' version '5.1.0' added to file 'C:\Foo\Cmdlet\SendGreeting\SendGreeting.c
sproj'.
info : Committing restore...
info : Writing assets file to disk. Path: C:\Foo\Cmdlet\SendGreeting\obj\project.assets.json
log  : Restored C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj (in 1.04 sec).
```

Figure 5.38: Adding the PowerShell package

In *step 7*, you use PowerShell and a here-string to create the source code file for your new cmdlet. This step produces no output.

In *step 3*, you create your new project. This step also creates a file called `Class1.cs`, which is not needed for this project. In *step 8*, which generates no output, you remove this unneeded file.

In *step 9*, you build the cmdlet, which produces output like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 9. Building the cmdlet
PS C:\Foo\Cmdlet\SendGreeting> dotnet build
Microsoft (R) Build Engine version 16.8.0+126527ff1 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  SendGreeting -> C:\Foo\Cmdlet\SendGreeting\bin\Debug\net5.0\SendGreeting.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.55
```

Figure 5.39: Building the cmdlet

In *step 10*, you use `Import-Module` to import the DLL, which produces no output. With *step 11*, you use `Get-Module` to discover the imported module (containing just a single command), which looks like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 11. Examining the module's details
PS C:\Foo\Cmdlet\SendGreeting> Get-Module SendGreeting

ModuleType Version    PreRelease Name           ExportedCommands
---------- -------    ---------- ----           ----------------
Binary     1.0.0.0               SendGreeting   Send-Greeting
```

Figure 5.40: Examining the imported module

In the final step in the recipe, *step 12*, you execute the cmdlet, which produces the following output:

```
PS C:\Foo\Cmdlet\SendGreeting> # 12. Using the cmdlet
PS C:\Foo\Cmdlet\SendGreeting> Send-Greeting -Name "Jerry Garcia"
Hello Jerry Garcia - have a nice day!
```

Figure 5.41: Using the cmdlet

There's more...

In *step 1*, you download the .NET SDK, which you need in order to create a cmdlet. There appears to be no obvious way to automate the installation, so you need to run the installer and click through its GUI in order to install the .NET SDK.

The .NET SDK should be relatively straightforward to install, as shown in this recipe. You download and run the .NET installer and are then able to use the tools to build a cmdlet. But in a few, thankfully rare, cases the installation may not work correctly and/or the installer may get confused. You may need to uninstall any other SDKs you have loaded or perhaps inquire on a support forum such as `https://community.spiceworks.com/programming/powershell`. That being said, you should not have this problem on a newly created VM.

In *step 3*, you create a new class library project. This step also creates the `SendGreeting.csproj` file, as you can see. This file is a Visual Studio .NET C# project file that contains details about the files included in your `Send-Greeting` cmdlet project, assemblies referenced from your code, and more. For more details on the project file, see `https://docs.microsoft.com/en-us/aspnet/web-forms/overview/deployment/web-deployment-in-the-enterprise/understanding-the-project-file`.

In *step 5*, you create the `global.json` file. If you do not create this file, the `dotnet` command will compile your cmdlet with the latest version of .NET Core loaded on your system. Using this file tells the project build process to use a specific version of .NET Core (such as version 5.0). For an overview to this file, see `https://docs.microsoft.com/dotnet/core/tools/global-json`.

In *step 9*, you compile your source code file and create a DLL containing your cmdlet. This step compiles all of the source code files contained in the folder to create the DLL. This means you could have multiple cmdlets, each in a separate source code file, and build the entire set in one operation.

In *step 11*, you can see that the module, `SendGreeting`, is a binary module. A binary module is one just loaded directly from a DLL. This is fine for testing, but in production, you should create a manifest module by adding a manifest file. You can get more details on manifest files from `https://docs.microsoft.com/en-us/powershell/scripting/developer/module/how-to-write-a-powershell-module-manifest?view=powershell-7.1`. You should also move the module and manifest, and any other module content such as help files, to a supported module location. You might also consider publishing the completed module to either the PowerShell gallery or to an internal repository.