5

Exploring the .NET Framework

In this chapter, we cover the following recipes:

* Exploring .NET Assemblies
* Examining .NET Classes
* Leveraging .NET Methods
* Creating a C# Extension
* Writing a simple cmdlet

# Introduction

Microsoft first launched .NET in June 2000, amidst 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 (later re-renamed Windows Server 2003), Visual Studio.NET and even MapPoint .NET. As is often the case, .NET provided features which were superseded by newer features based on advances in technology. For example, SOAP (Simple Object Access Protocol) and XML-based web services have given way to REST (Representation State Transfer), and JSON (JavaScript Object Notation) and SOAP are not a part of .NET 5.0.

Microsoft made considerable improvements to .NET over time. Each release of .NET added new features based on customer feedback. .NET started as closed-source, but Microsoft has transitioned .NET to open source. PowerShell 7.1 is built on top of and takes advantage of the latest version of .NET Core, .NET 5.0.

Beyond marketing, .NET is many things at once. It's an Application Program Interface (API), a programming model with associated tools and a new run-time, as well as a new style of architecture utilizing XML Web Services. .NET is an environment in which developers can develop rich applications and web sites. Details of the internals of .NET are relevant for developers building .NET desktop applications or ASP.NET web solutions. But for most IT Professionals, these are interesting but not directly relevant.

For the IT Professional, .NET provides the underpinnings of PowerShell but is otherwise mostly transparent. You use cmdlets which in turn use .NET to carry out their work. Additionally, PowerShell allows you to use the features of .NET directly. Here is a high-level illustration of the components of .NET:



**Insert image B42024\_05\_01.png**

The key components of .NET 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 support .NET 5.0, the basis for PowerShell 7.1, on Windows, Linux, and the Apple Mac. For a full list of supported operating systems, 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 delivers a managed execution environment as well as type safety and code access security. The CLR manages memory, threads, objects, resource allocation/de-allocation, and garbage collection. The CLR also contains the Just-In-Time (JIT) compiler and the Class Loader (responsible for loading classes into an application at run time).
    - **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. The BCLs are run-time DLLs containing the .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, and you can call them directly.
    - **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.
    - **Languages** - this is the language that a developer uses to develop PowerShell cmdlets. You can use an extensive variety of languages, based on your preferences and skillset, although most cmdlet developers use 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.

Every operating system implements essential computing functions such as I/O, networking and much, much more. The .NET Framework sits on top of the OS and makes use of the OS. The BCLs are .NET’s APIs. And provide a huge range of services to applications. Application developers write applications utilizing the classes and types within these base class libraries.

The CLR is the environment in which all .NET applications, including PowerShell, run. The CLR provide the application developer with a range of facilities and approaches to application development. Still, for the IT Pro, the inner workings of the CLR are not all that important (usually!). For an overview of the CLR, see https://docs.microsoft.com/dotnet/standard/clr/

Win32 is the core Windows API set. Before .NET, developers wrote desktop applications using Win32. The Component Object Model (COM) and Distributed COM (DCOM) were object-oriented approaches to application development. WMI represented another set of APIs that developers use in developing applications. .NET provides a mechanism for an application to access WIN32, COM, and WMI in addition to the base BCL.

For most IT Pros, the language used to develop PowerShell and PowerShell extensions is not directly relevant. With that said, the PowerShell language is broadly compatible with C#. You can describe PowerShell as on the glide scope to C#.

At the top of this stack is PowerShell itself. PowerShell language constructs and syntax make use of the components of the .NET Framework and utilize classes and types in the Base Class libraries.

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 a simple cmdlet.

# Exploring .NET Assemblies

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

The output of the .NET Language compiler is not native machine code (as would be the casefor, say, C++. Instead, comp0ilers create Intermediate Language (IL) output which is platform-independent. At run time, .NET loads the assemblies needed by an application and compiles this IL code into native machine code. This approach means that .NET can use the same DLL on both 32-bit and 64-bit versions of Windows. The Just-In-Time compilation process could induce a performance hit due to compiling lots of assembles each time you run an application or load a .NET DLL. To avoid this performance issue, .NET enables a developer to pre-compile assemblies in the global assembly cache.

Most PowerShell modules and most PowerShell commands are implemented as one or more assemblies. When PowerShell loads a module, the module manifest lists the assemblies which make up the module. For example, the Microsoft.PowerShell.Management module provides many core PowerShell commands such as Get-ChildItem or Get-Process. A module's manifest lists a nested module (i.e Microsoft.PowerShell.Commands.Management.dll) which is the assembly containing the actual commands.

A great feature of PowerShell is the ability to invoke directly a .NET Class method or to obtain a static .NET Class value. You can observe the syntax for calling a .NET Method or a .NET field in the recipes in this chapter. In general, with this syntax, you enclose the class name in square brackets, and follow it with two “:” characters and the name of the method as you see below.

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 Next, you 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 = [appdomain]::CurrentDomain.GetAssemblies()

"Assemblies loaded: {0:n0}" -f $Assemblies.Count

1. Viewing first 10

$Assemblies | Select-Object -First 10

1. Checking assemblies in Windows PowerShell

$SB = {

  [appdomain]::CurrentDomain.GetAssemblies()

}

$PS51 = New-PSSession -UseWindowsPowerShell

$Assin51 = Invoke-Command -Session $PS51 -ScriptBlock $SB

"Assemblies loaded in Windows PowerShell: {0:n0}" -f $Assin51.Count

1. Viewing Microsoft.PowerShell assemblies

$Assin51 |

  Where-Object FullName -Match "Microsoft\.Powershell" |

    Sort-Object -Property Location

1. Exploring the Microsoft.PowerShell.Management module

$MOD = Get-Module -Name Microsoft.PowerShell.Management -ListAvailable

$MOD  | Format-List

1. Viewing module manifest

$MANIFEST = Get-Content -Path $MOD.Path

$MANIFEST | Select-Object -First 20

1. Discovering the module's assembly

Import-Module -Name Microsoft.PowerShell.Management

$MATCH = $MANIFEST | Select-String Modules

$LINE = $MATCH.Line

$DLL = ($Line -Split '"')[1]

Get-Item -Path $PSHOME\$DLL

1. Viewing associated loaded assembly

$Assemblies2 = [appdomain]::CurrentDomain.GetAssemblies()

$Assemblies2 | Where-Object Location -match $DLL

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

$COMMANDS  = $ASSEMBLIES2

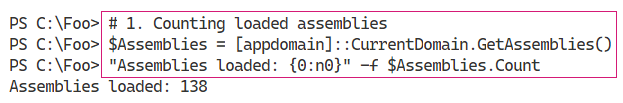
               Where-Object Location -match Commands.Management\.dll

$COMMANDS.GetTypes() |

  Where-Object Name -match "Addcontentcommand$"

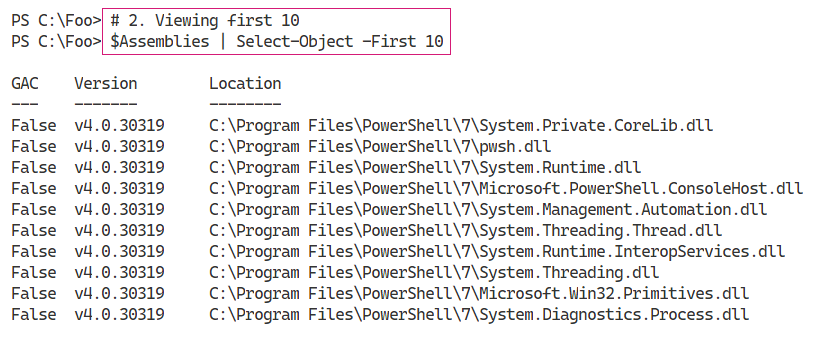
## How it works...

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:



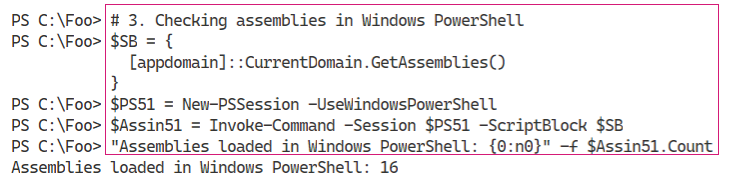
Insert image B42024\_05\_02.png

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



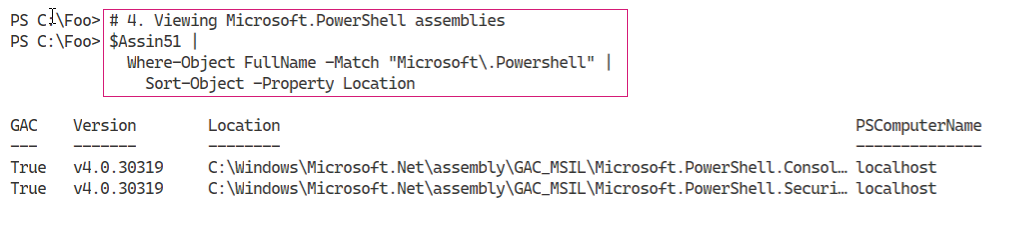
Insert image B42024\_05\_03.png

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



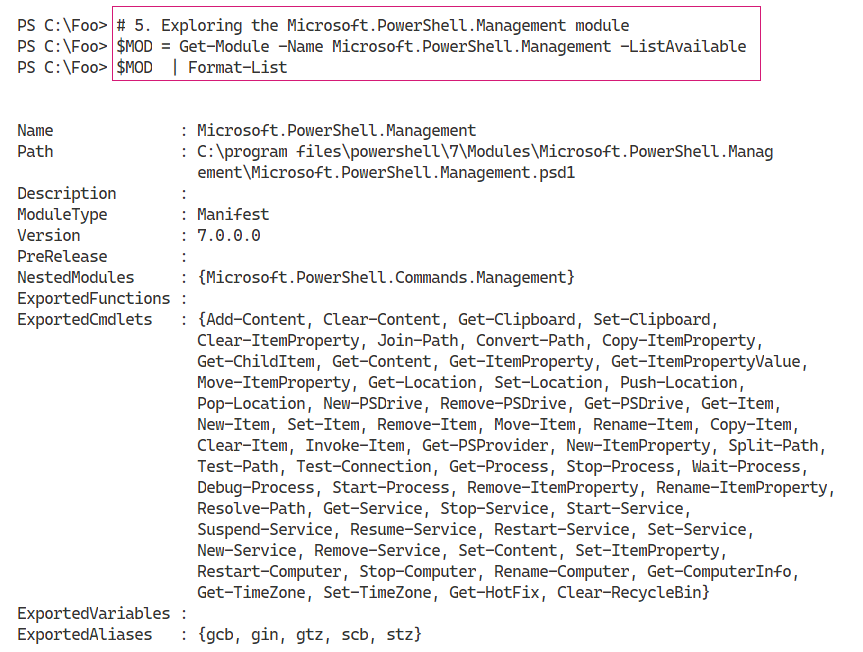
Insert image B42024\_05\_04.png

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



Insert image B42024\_05\_05.png

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



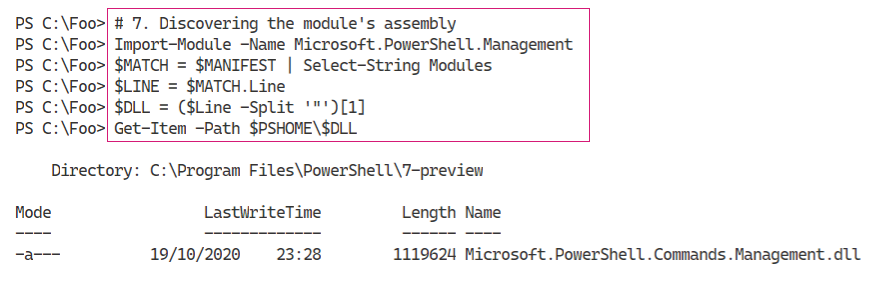
Insert image B42024\_05\_06.png

In step 6, you view the manfest for the Microsoft.PowerShell.Management module. The figure below shows the first 20 lines of the manifest, which looks like this:



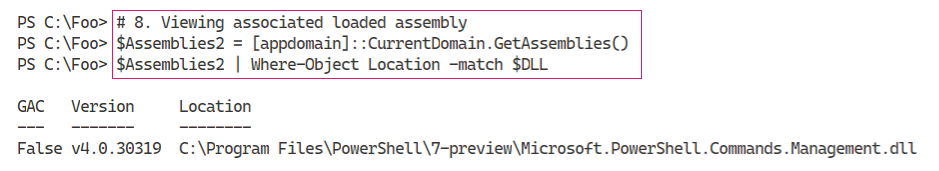
Insert image B42024\_05\_07.png

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:



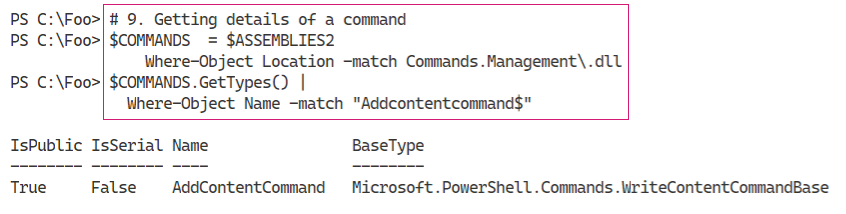
Insert image B42024\_05\_08.png

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



Insert image B42024\_05\_09.png

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



Insert image B42024\_05\_10.png

## There's more...

In this recipe, you have seen the .NET assemblies used by PowerShell. These consist of both the .NET Framework assemblies, but also the assemblies which implement PowerShell commands. For example, the Add-Content cmdlet is found in the Microsoft.PowerShell.Management module.

In step 1, you use the System.Appdomain class’s 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 step 3 and step 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 module manifest for the Microsoft.PowerShell.Management module. The output cuts off the complete list of cmdlets exported by the module and the (very long) digital signature for the module manifest. In this figure, 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 alias the Add-Content cmdlet and in step 8, you can see that the assembly is loaded. In step 9, you can 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, and you can read the source code for this cmdlet.

# Examining .NET Classes

With .NET, a class defines a type of object. Objects 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 class System.IO.FileInfo and/or System.IO.DirectoryInfo.

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: class and type. A type defines a simple object that can live, at runtime on your CPUs’ stack. Classes, being more complex, always live in the global heap. The global heap is a large area of memory which .NET/PowerShell use to hold typically short-lived based objects. 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. In almost all cases, the difference between type and class is probably not overly relevant to IT Professionals using PowerShell.

The garbage collection process is also not important for the IT Professional in most cases. The scripts you see in this book, for example, are not 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/.

But there are cases where the GC process can impact on performance. For example, the System.Array class creates objects of fixed size. if your script adds an item to an array, .NET creates a totally new copy of the array (plus the addition), and removes the old one. If you are adding a few iterms to the array, the performance hit is negligible. But if you are adding millions of occurrences, the performance can suffer. To avoid this, you can just use the ArrayList class which supports adding/removing items from an array without a huge performance penalty.

PowerShell 7,1 is built on .NET 5.0 which featured a number of improvements to 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 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 “Leveraging .NET Methods”. With .NET An event is something that can happen to an object occurrence, such when an evemt generated when a Windows process has completed. .NET events are not covered in this book, although using WMI events is described in Chapter 16 in the “Using WMI Events” recipe.

PowerShell also implements an Extended Type System (ETS). The ETS enables you to extend a .NET class on the fly. Cmdlet developers can add a variety of properties to objects, including a script property and an alias property. The script property appears to be a regular property, except at runtime, PowerShell runs a script to create the property value. An alias property is an alternate name for an objects existing property. For more on the ETS, see https://docs.microsoft.com/powershell/scripting/developer/ets/overview.

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 also have static fields and static methods. Static fields/methods are aspects of the class of a whole as opposed to a specific class instance. A static field 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 that is related to the class, such as parsing a string to determine if it is a valid 32-bit integer.

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

1. Discovering the underlying class

$TYPE = $FILE.GetType().FullName

".Net Class name: $TYPE"

1. Getting member types of Fileinfo object

$File |

  Get-Member |

    Group-Object -Property MemberType |

      Sort-Object -Property Count -Descending

1. Discovering the underlying type of an integer

$I = 42

$IntType  = $I.GetType()

$TypeName = $IntType.Name

$BaseType = $IntType.BaseType.Name

".Net Class name      : $TypeName"

".NET Class Base Type : $BaseType"

1. 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

1. Looking at static values within a class

$Max = [Int32]::MaxValue

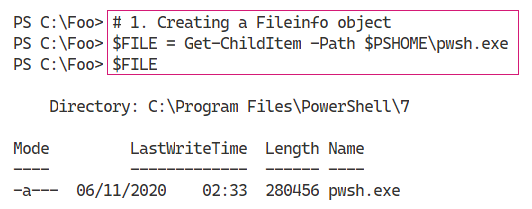
$Min = [Int32]::MinValue

"Minimum value [$Min]"

"Maximum value [$Max]"

## How it works...

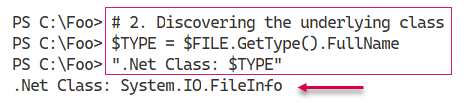
In step 1, you use the Get-ChildItem cmdlet to return an object representing the PowerShell 7 executable file, with output like this:



Insert image B42024\_05\_11.png

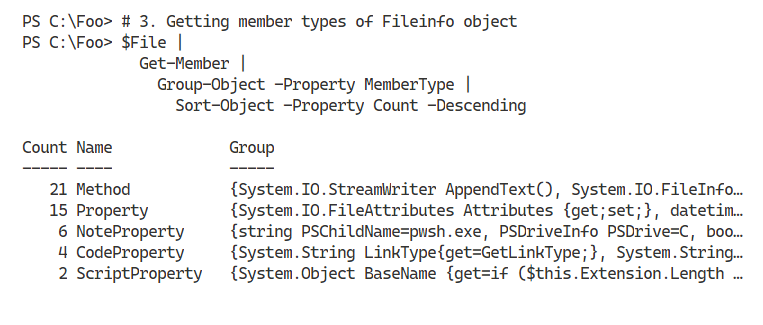
You can determine the class name of an object by using the GetType() method. This method is present on every object and returns information about the object’s type.

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



Insert image B42024\_05\_12.png

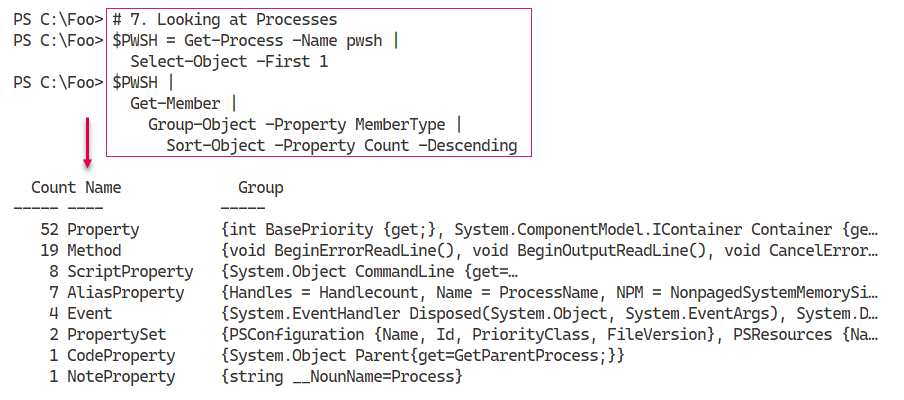
In step 3, you use Get-Member to display the different member types contained within a FileInfo object, with output like this:



Insert image B42024\_05\_13.png

In step 4, you use the GetType() method, present on all .NET objects, to return information about the $IntType’s underlying type. The output looks like this.

Insert image B42024\_05\_14.png



## 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 the 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 favorite 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 classnames in your scripts, a best practice is to spell out the full classname.

# Leveraging .NET Methods

With .NET a method is some action 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 cmdlets and only dip down into using .NET classes and methods directly.

.NET methods can be very useful in order to perform some operation which has no PowerShell cmdlet. One use of a .NET method is 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. 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 objects Kill() method. Of course, to help IT Professionals, the PowerShell team created the Kill alias to the Stop-Process cmdlet.

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

At present 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

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. 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 Kill. PowerShell then calls the objects Kill() method. In practice, it’s 4 characters to type vs 11 (or is you are using tab completion to best effect 8), 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.

As you saw in Examining .NET Classes”, you can pipe any object to the Get-Member cmdlet to discover the methods for any 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

1. Obtaining methods on the Notepad process

$Notepad = Get-Process -Name Notepad

$Notepad | Get-Member -MemberType method

1. Using the Kill() method

$Notepad |

  ForEach-Object {$\_.Kill()}

1. Confirming Notepad process is destroyed

Get-Process -Name Notepad

1. 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"

}

1. Viewing files in $Path folder

$Files = Get-ChildItem -Path $Path

$Files | Format-Table Name, Attributes

1. Encrypting the files

$Files| ForEach-Object Encrypt

1. Viewing file attributes

Get-ChildItem -Path $Path |

  Format-Table -Property Name, Attributes

1. Decrypt and view 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 examind the methods available to you. The output looks like this:

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:

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:

In step 7, you use the encrypt method to encrypt the files, generating no output. In step 8, you re-view the files attributes, which looks like this:

Finally, with step 9, you decrypt the files using the Decrypt() method, which generates no output. You can see, in the output of step 10, that the files are no longer encrypted. The output of this step looks like this:

## There's more...

The output from step 1

# Creating a C# Extension

## Getting Ready

You run this recipe on

## How to do it...

1. Displaying counts of available PowerShell commands

## How it works...

In step 1,

Insert image B42024\_05\_01.png

## There's more...

The output from step 1

# Writing a simple cmdlet

## Getting Ready

You run this recipe on

## How to do it...

1. Displaying counts of available PowerShell commands

## How it works...

In step 1,

Insert image B42024\_05\_0`.png

## There's more...

The output from step 1