


POWERSHELL

Grupo ANA

A series of several thin, parallel diagonal lines in a light blue color, extending from the bottom left towards the top right of the page.

Alejandro Álvarez Corujo
Javier Escribano Salgado
David Franco Orol
Lukas Gómez Evangelista
Adrián Nieto Antelo
Diego Villodas Zapata

1 Contenido

1	Contenido.....	1
2	Introducción.....	3
2.1	Versiones	4
3	Variables y tipos de datos.....	6
3.1	Variables creadas por el usuario.....	6
3.1.1	Constantes	8
3.2	Variables predefinidas	8
3.2.1	Variables automáticas	8
3.2.2	Variables de preferencia	9
3.3	Tipos de datos.....	10
4	Ejecución condicional y bucles.....	11
4.1	Bucles	11
4.1.1	Bucle While.....	12
4.1.2	Bucle Do-While	12
4.1.3	Bucle Do-Until	13
4.1.4	Bucle For.....	13
4.1.5	Bucle Foreach	13
4.2	Estructuras condicionales	14
4.2.1	<i>If, Else, Elseif</i>	14
4.2.2	Switch	14
5	Funciones.....	15
5.1	Nomenclatura.....	15
5.2	Funciones Simples.....	16
5.3	Parámetros	17
5.4	Funciones Avanzadas	18
5.5	Tratamiento de Errores	22
6	Cmdlets.....	23
7	Objetos.....	25
7.1	Pipelines	28
8	Módulos.....	30
8.1	Componentes y tipos de módulo	30
8.1.1	Módulos de script (Script).....	31
8.1.2	Módulos binarios (Binary)	32
8.1.3	Módulos dinámicos (Dynamic).....	32

8.1.4	Módulos del manifiesto (Manifest)	33
8.2	Añadir un módulo	34
8.3	Importar un módulo.....	35
8.4	Eliminar un módulo.....	36
9	Interacción con Windows Management Instrumentation (WMI)	37
9.1	Introducción.....	37
9.2	Obtener un objeto de WMI.....	38
9.3	Enumerar clases WMI	38
9.4	Obtener información detallada de las clases de WMI.....	39
9.5	Visualizar prop. no predeterminadas con cmdlets de formato	40
10	Bibliografía.....	41

2 Introducción

PowerShell es un shell de comandos moderno que nace en 2006 de la mano de Microsoft y su inventor Jeffrey Snover, el cual lo define como “**Eficiente**, extensible y con una **curva de aprendizaje poco pronunciada**”. PowerShell es a la vez un intérprete de comandos y potente lenguaje de scripts que incluye las mejores características de otros shells populares. A diferencia de la mayoría de los shells que solo aceptan y devuelven texto, PowerShell acepta y devuelve objetos .NET, y saca una de sus principales fortalezas gracias al Framework .NET en el que se apoya. El Framework .NET, es una biblioteca de clases inmensa a partir de la cual crearemos objetos; objetos los cuales nos van a permitir actuar sobre el conjunto del sistema operativo con un mínimo esfuerzo. Aquí podemos ver otra de las grandes características de PowerShell, y es que es un lenguaje orientado a objetos.

Con PowerShell no se trabajará únicamente con texto, como es el caso con la mayoría de las demás shells, sino con **objetos**. Por ejemplo, cuando use una pipe «|» para pasar datos a un comando, no transmitirá texto, sino un objeto con todo lo que lo caracteriza (sus propiedades y sus métodos). Gracias a esto es cómo los scripts PowerShell son, por lo general, más concisos que en los demás lenguajes.

Como lenguaje de scripting, PowerShell se usa normalmente para **automatizar la administración de sistemas**. También se usa para compilar, probar e implementar soluciones, a menudo en entornos de CI/CD (Integración continua / Desarrollo continuo). Todas las entradas y salidas son objetos de .NET. No es necesario analizar la salida de texto para extraer información de la salida. El lenguaje de scripting de PowerShell incluye las siguientes características:

- Extensible mediante funciones, clases, scripts y módulos.
- Sistema de formato extensible para una salida fácil.
- Sistema de tipos extensible para crear tipos dinámicos.
- Compatibilidad integrada con formatos de datos comunes, como CSV, JSON y XML.
- Uso de módulos, para dividir y organizar el código en unidades propias reutilizables y compartibles.

A nivel de shell incluye las siguientes características:

- Un historial de línea de comandos sólido.
- Finalización con tabulación y predicción de comandos.
- Admite alias de comando y parámetro.
- Canalización para encadenar comandos.
- Sistema de ayuda en la consola, similar a las páginas *man* de UNIX.

Por otra parte, PowerShell posee un juego de comandos extremadamente rico y que no deja de crecer con cada nueva versión. Desde la versión 3 contábamos ya con aproximadamente más de doce veces el número de comandos del intérprete CMD.exe. Dicho esto, los comandos CMD pueden usarse desde PowerShell, si existe la necesidad. Los comandos PowerShell

poseen la inmensa ventaja de estar todos diseñados en base al mismo modelo (heredan todos del mismo tipo de objetos). Tienen nombres fáciles de retener, y es fácil adivinar los nombres de comandos que no conocemos. Cada uno de ellos posee además una (o varias) serie de parámetros, parámetros que memorizaremos gracias a la coherencia existente en conjunto.

2.1 Versiones

Las versiones de PowerShell se suelen presentar con cada lanzamiento de un sistema operativo. Se puede saber la versión que se tiene de PowerShell usando el comando **\$PSVersionTable**

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

Name                           Value
----                           -
PSVersion                      5.1.19041.906
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.19041.906
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1

PS C:\Users\adria>
```

A mayo de 2021 hay documentadas las siguientes:

Versión 1.0

Fue presentada en 2006 y se lanzó para Windows **XP** SP2, Windows Server 2003 SP1 y Windows **Vista**, además de ser un componente opcional de Windows Server 2008. Esta versión aumenta las capacidades del CMD.exe y **cubre varias necesidades que la consola clásica no**. Entre las novedades permite el manejo de objetos, lo que aumenta la semántica del lenguaje de scripting y la interacción con la máquina, y los *cmdlets*, para realizar tareas administrativas.

Versión 2.0

Fue presentado en 2009 y se lanzó para **Windows 7** y Windows Server 2008 R2, en los que ya está integrado, y para Windows XP con Service Pack 3, Windows Server 2003 con Service Pack 2 y Windows Vista con Service Pack 1. Esta versión **añade un entorno gráfico** (PowerShell ISE), la interacción remota con uno o más equipos, módulos para el desarrollo de código, más *cmdlets* y **ejecución en segundo plano de los trabajos**.

Versión 3.0

Fue presentado en 2012 y se lanzó para **Windows 8** y Windows Server 2012. Entre sus novedades podemos encontrar los workflows, que permiten realizar tareas administrativas más grandes y complejas; sesiones sólidas, para recuperarse automáticamente de errores de ejecución o de red; tareas programadas; se simplificó el lenguaje y se añade el *cmdlet* **Show-Command**.

Versión 4.0

Fue presentado en 2013 y se lanzó para **Windows 8.1** y Windows Server 2012 R2. Entre sus novedades podemos apreciar el *Desired State Configuration* (DSC), que incluye extensiones de lenguaje declarativo y herramientas que permitan el despliegue y la gestión de los datos de configuración para servicios de software, y la gestión del entorno en el que estos servicios se ejecutan; **Save-Help**, para que se pueda guardar la ayuda para los módulos que están instalados en equipos remotos; ISE mejorado con debugging y bugfix; **PipelineVariable switch**; diagnóstico de la red y se añaden el ForEach y el Where.

Versión 5.0

Fue presentado en 2015, pero por un bug se lanzó en 2016 para **Windows 10**. Esta versión define nuevas clases de PowerShell, aparecen los PowerShell .NET Enumerados, la depuración de la ejecución en los procesos remotos, la depuración de procesos en segundo plano y mejora el DSC. Esta es la última versión que usa el .NET Framework, que era de donde sacaba toda su funcionalidad PowerShell. **A partir de aquí se usa la versión .NET Core, que es open source y multiplataforma.**

Versión Core 6.0

Fue presentada en 2016, pero se lanzó en 2018. **Es la primera versión open source y multiplataforma de PowerShell.** Esta versión permite la instalación simultanea de varias versiones y mejora el rendimiento, la codificación de archivos de texto y las interacciones con los servicios web.

Versión Core 6.1

A partir de esta versión se comenzó a lanzar versiones cada seis meses. Esta mejora más el rendimiento y **proporciona soporte para las últimas versiones de Windows, MacOS y Linux.**

Versión Core 6.2

Esta versión mejora el rendimiento, corrige errores y proporciona otras mejoras menores de *cmdlet* e idiomas.

Versión 7

Es la última versión hasta la fecha. Entre sus nuevas características podemos encontrar el `ForEach-object -parallel`, una nueva vista de error y cmdlet `Get-Error`, operadores `&&` y `||`, asignación nula y operadores de coalescencia nula (`??=` y `??`), y comandos `cmdlet` GUI heredados como `Out-GridView`.

3 Variables y tipos de datos

En PowerShell, las variables se representan mediante cadenas de texto que comienzan por el signo del dólar (\$).

Los nombres de variable **no distinguen mayúsculas de minúsculas** y pueden incluir espacios y caracteres especiales. Sin embargo, los nombres de variable que incluyen caracteres especiales y espacios son difíciles de usar y deben evitarse.

De forma predeterminada, las variables solo están disponibles en el ámbito en el que se crean.

Por ejemplo, una variable que se crea en una función solo está disponible dentro de la función. Una variable que se crea en un script solo está disponible en el script.

Las variables pueden ser de varios tipos, las que vamos a ver son las siguientes:

- Variables creadas por el usuario
- Variables predefinidas

3.1 Variables creadas por el usuario

Una curiosidad es que las variables no debemos declararlas ni inicializarlas antes de utilizarlas, y asignarle un tipo se hace de forma automática, aunque también podremos hacerlo de forma manual, para crear una variable con un valor asignado basta con poner el `$nombrevariable=valor`.

```
PS C:\Users\Usuario> $palabra="hola "  
PS C:\Users\Usuario> $frase="estamos aprendiendo el uso de las variables"  
PS C:\Users\Usuario> $numero1=2  
PS C:\Users\Usuario> $numero2=4  
PS C:\Users\Usuario> ${caracteres especiales}="Esto son caracteres especiales"
```

Ejemplos de creación de variables por el usuario

Como podemos observar hemos creado varias variables sencillas, también hicimos una variable con caracteres especiales, las variables cuyo nombre contenga caracteres especiales o espacios se deben crear entre corchetes, en este caso tiene un espacio en blanco el cuál no podría ser permitido sino tuviese corchetes.

Ahora mostraremos el uso básico de las variables:

Se mostrarán las variables con el nombre asignado precedido del símbolo del dólar como se muestra en la imagen:

```
PS C:\Users\Usuario> $frase
estamos aprendiendo el uso de las variables

PS C:\Users\Usuario> $numero1
2

PS C:\Users\Usuario> ${caracteres especiales}
Esto son caracteres especiales
```

Haremos unas operaciones básicas de suma y concatenación:

```
PS C:\Users\Usuario> $palabra+ $frase
hola estamos aprendiendo el uso de las variables

PS C:\Users\Usuario> $palabra+ $numero1
hola 2

PS C:\Users\Usuario> $palabra+ ${caracteres especiales}
hola Esto son caracteres especiales

PS C:\Users\Usuario> $numero1 +$numero2
6

PS C:\Users\Usuario> $numero1 -$numero2
-2
```

Se podrá borrar el contenido de una variable con:

```
PS C:\Users\Usuario> Clear-Variable -Name palabra
PS C:\Users\Usuario> $palabra
```

Se podrá borrar la variable con:

```
PS C:\Users\Usuario> Remove-Variable -Name frase
PS C:\Users\Usuario> $frase|
```

Ahora mostraremos como las variables también pueden almacenar a los *arrays* y haremos algunas operaciones sobre ellos:

En la imagen a continuación, podemos ver la creación del array, después mostraremos su longitud, el primer elemento asignado, a continuación, le añadimos un *char* y vemos que se añadió correctamente, por último, elegimos los elementos seleccionados en la imagen y vemos que quedaría modificado el array.


```

PS C:\Users\Usuario> $array="tres", "dos","uno"
PS C:\Users\Usuario> $array.Length
3
PS C:\Users\Usuario> $array[0]
tres
PS C:\Users\Usuario> $array+="cero"
PS C:\Users\Usuario> $array
tres
dos
uno
cero
PS C:\Users\Usuario> $array=($array[0],$array[1],$array[3])
PS C:\Users\Usuario> $array
tres
dos
cero

```

3.1.1 Constantes

Se podrá crear una constante con el siguiente comando:

```

PS C:\Users\Usuario> New-Variable nombre -Option Constant -Value 10
PS C:\Users\Usuario> New-Variable nombre1 -Option ReadOnly -Value 10

```

También se podría crear con el segundo comando de la línea anterior dado de que si es de solo lectura tampoco podríamos modificar el valor.

```

PS C:\Users\Usuario> $nombre=30
No se puede sobrescribir la variable nombre porque es de solo lectura o
constante.
En línea: 1 Carácter: 1
+ $nombre=30
+ ~~~~~
+ CategoryInfo          : WriteError: (nombre:String) [],
SessionStateUnauthorizedAccessException
+ FullyQualifiedErrorId : VariableNotWritable

```

Como podemos observar nos da **un error al intentar cambiar el valor de la constante** y tampoco nos especifica si es una constante o que solo tiene permisos de lectura.

3.2 Variables predefinidas

Este tipo de variable se divide en dos:

3.2.1 Variables automáticas

Este tipo de variable almacena el estado del PowerShell. Haremos ejemplos de este tipo de variable:

Existe una variable que nos indica si la última operación se ejecutó correctamente:

```

PS C:\Users\Usuario> $?
False

```

Nos devuelve falso debido a que la operación de cambiar el valor de una constante nos dio un fallo, en cambio, si ejecutamos este mismo comando ahora, el resultado será **True**.

```
PS C:\Users\Usuario> $?  
True
```

En efecto, la respuesta es **True**, como habíamos anunciado.

Por otra parte, podemos ver el último error cometido:

```
PS C:\Users\Usuario> $Error[0]  
No se puede sobrescribir la variable nombre porque es de solo lectura o  
constante.  
En línea: 1 Carácter: 1  
+ $nombre=30  
+ ~~~~~  
+ CategoryInfo          : WriteError: (nombre:String) [],  
SessionStateUnauthorizedAccessException  
+ FullyQualifiedErrorId : VariableNotWritable
```

Para ver la ruta de acceso que representa la ruta de acceso completa de la ubicación de directorio actual del espacio de ejecución actual de PowerShell.

```
PS C:\Users\Usuario> $pwd  
  
Path  
----  
C:\Users\Usuario
```

Para ver el último token en la última línea recibida por la sesión:

```
PS C:\Users\Usuario> $$  
$pwd
```

3.2.2 Variables de preferencia

Variables que personalizan el comportamiento de PowerShell. Haremos ejemplos de este tipo de variable:

Existe una variable que determina el formato de presentación de los mensajes de error en PowerShell.

Su estado es **NormalView**, y podremos cambiarlo a **CategoryView** para cambiar el formato del error:

```
get-childitem nofile.txt  
get-childitem : No se encuentra la ruta de acceso  
'C:\Users\Usuario\nofile.txt' porque no existe.  
En línea: 1 Carácter: 1  
+ get-childitem nofile.txt  
+ ~~~~~  
+ CategoryInfo          : ObjectNotFound:  
(C:\Users\Usuario\nofile.txt:String) [Get-ChildItem], ItemNotFoundException
```

```
+ FullyQualifiedErrorId :
PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

PS C:\Users\Usuario> $ErrorView = "CategoryView"

PS C:\Users\Usuario> get-childitem nofile.txt
ObjectNotFound: (C:\Users\Usuario\nofile.txt:String) [Get-ChildItem],
ItemNotFoundException
```

\$FormatEnumerationLimit determina el número de elementos enumerados que se incluyen en una pantalla:

```
$FormatEnumerationLimit
4

PS C:\Users\Usuario> Get-Service | Group-Object -Property Status

Count Name Group
-----
154 Stopped {AarSvc_83eee11, AJRouter, ALG, AppIDSvc...}
128 Running {AdobeARMSservice, Appinfo, AppXSvc,
AudioEndpointBuilder...}
```

Al cambiar el valor de 4 a 10, veremos cómo nos muestra 10 resultados en el campo Group

```
PS C:\Users\Usuario> $FormatEnumerationLimit = 10

PS C:\Users\Usuario> Get-Service | Group-Object -Property Status

Count Name Group
-----
154 Stopped {AarSvc_83eee11, AJRouter, ALG, AppIDSvc,
AppReadiness, autotimesvc, AxInstSV, BcastDVRUserService_83eee11, BDESVC,
BEService...}
128 Running {AdobeARMSservice, Appinfo, AppXSvc,
AudioEndpointBuilder, Audiosrv, BFE, BITS, BrokerInfrastructure, BTAGService,
BthAvctpSvc...}
```

3.3 Tipos de datos

La tabla de datos siguiente muestra los tipos de datos más utilizados, aunque existan más:

Tipo de dato	Descripción
[string]	Cadena de caracteres
[char]	Un solo carácter Unicode
[int]	Entero con signo de 32 bits
[double]	Numero en coma flotante 64 bits
[datetime]	Fecha y hora
[bool]	Valor lógico
[array]	Conjunto de valores

El tipo de variable de PowerShell **se establece automáticamente en función del valor que le asigne**, pero **también puede asignar el tipo manualmente**, esto es útil cuando trabajamos con muchas variables para no poner incorrectamente ningún tipo.

Se asigna manualmente poniendo el tipo de dato antes de la variable, por tanto, no aceptará otro valor que no sea del tipo indicado:

```
PS C:\Users\Usuario> [datetime]$fecha="09/10/2021"
PS C:\Users\Usuario> [datetime]$fecha
viernes, 10 de septiembre de 2021 0:00:00

PS C:\Users\Usuario> [datetime]$fecha=""
MetadataError: (:) [], ArgumentTransformationMetadataException
```

```
PS C:\Users\Usuario> [int]$numero=0
PS C:\Users\Usuario> [int]$numero="cero"
MetadataError: (:) [], ArgumentTransformationMetadataException
```

Para ver el tipo de una variable podremos hacerlo a través de **Get-Type**:

```
PS C:\Users\Usuario> $fecha.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     DateTime                                     System.ValueType

PS C:\Users\Usuario> $numero.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     Int32                                       System.ValueType
```

Para **solo mostrar el nombre** del tipo de dato, sin información adicional se hace con:

```
PS C:\Users\Usuario> $fecha.GetType().Name
DateTime
```

Al convertir un **double** a un **int**, redondea el número:

```
PS C:\Users\Usuario> [float]$flotante=8.6
PS C:\Users\Usuario> [int]$flotante
9
```

4 Ejecución condicional y bucles

Como PowerShell permite es una interfaz de consola que permite ejecutar scripts también permite como consecuencia la ejecución condicional y de bucles, ya que se trata de algo básico en la mayoría de los lenguajes de programación.

4.1 Bucle

Como su propio nombre indica los **bucles son estructuras repetitivas**. En programación estos permiten la ejecución repetitivamente de varias instrucciones que se encuentren dentro del bucle.

4.1.1 Bucle While

En los bucles *while* las instrucciones que se encuentren dentro del mismo se ejecutan mientras una condición a evaluar sea verdadera. Esto lo logra ya que primero evalúa la condición, y si esta es falsa no ejecuta el bloque de instrucciones, mientras que si es verdadera si se ejecuta este bloque. Esto se puede apreciar en el siguiente ejemplo:

```
PS C:\Users\adria> $a = 1
PS C:\Users\adria> $b = 1
PS C:\Users\adria> while ($a -ne 5) {Write-Host $a; $a++}
1
2
3
4
PS C:\Users\adria> while ($b -ne 5) {$b++; Write-Host $b}
2
3
4
5
PS C:\Users\adria>
```

Bucles *while* que se ejecutan mientras **\$a** y **\$b** (respectivamente) sea menor que 5

4.1.2 Bucle Do-While

Es una **variante** del bucle **while**. Este, a diferencia del anterior, primero ejecuta el bloque de instrucciones y al final comprueba la condición, lo que permite que siempre se ejecute al menos una vez el bloque. En el siguiente ejemplo no se aprecia diferencia con el bucle *while*, ya que al cumplirse la condición de primeras no se ejecuta el bucle igual que si fuera un *while*.

```
PS C:\Users\adria> $a = 1
PS C:\Users\adria> $b = 1
PS C:\Users\adria> do {$a++; Write-Host $a} while ($a -ne 5)
2
3
4
5
PS C:\Users\adria> do {Write-Host $b; $b++;} while ($b -ne 5)
1
2
3
4
PS C:\Users\adria>
```

Bucles *do-while* que se ejecutan mientras **\$a** y **\$b** (respectivamente) sea menor que 5

Mientras tanto, en el siguiente ejemplo el bloque de instrucciones se ejecuta 1 vez al evaluarse la condición a posteriori, cosa que no sucedería en el bucle *while* ya que la condición no se cumple.

```
PS C:\Users\adria> $a = 1
PS C:\Users\adria> do {Write-Host $a; $a++} while ($a -eq 0)
1
PS C:\Users\adria>
```

4.1.3 Bucle Do-Until

El bucle *do-until* es otra variante del *while*, que funciona de la misma manera que el *do-while* (evaluando la condición al final), con la diferencia que el bloque **se repite mientras que la condición sea falsa**, al contrario del *do-while*, donde el bloque se repetía mientras la condición fuera verdadera.

```
PS C:\Users\adria> $a = 1
PS C:\Users\adria> do {$a++; Write-Host $a} until ($a -eq 5)
2
3
4
5
PS C:\Users\adria>
```

Bucle *do-until* que se ejecuta mientras **\$a** no sea igual o mayor que 5

Como vemos en, lo único que cambia con respecto al bucle *do-while* es que en lugar de comprobar que **a** no es igual a 5 aquí se comprueba que sea igual a 5 para que de la misma salida.

4.1.4 Bucle For

El bucle *for* permite **realizar el bloque de instrucciones un número determinado de veces**. Esto sucede porque cuando usamos un bucle *for* le indicamos un valor de inicio, un incremento (cuanto va a aumentar el valor de inicio cada vez que se ejecute el bloque) y una condición de repetición. El bucle *for* funciona de la siguiente manera: Primero se evalúa el valor de inicio (generalmente suele ser la asignación de un valor a una variable), de segundo se evalúa la condición de repetición, y si esta es falsa se termina el bucle, mientras que si es verdadera se ejecuta el bloque. Por último, tras la ejecución del bloque se incrementa el valor inicial en función del valor elegido de incremento y se vuelve a evaluar la condición de repetición.

```
PS C:\Users\adria> for ($i = 1;$i -le 10;$i++) {Write-Host $i}
1
2
3
4
5
6
7
8
9
10
PS C:\Users\adria>
```

Bucle *for* que se ejecuta mientras **\$i** no sea mayor que 10

4.1.5 Bucle Foreach

Es el mejor **para recorrer colecciones de datos**, ya que nos es necesario determinar cómo se avanza sobre los elementos de una colección.

Tiene dos maneras de usar, mediante *foreach*, en la que se usa una variable que solo existe dentro del bucle que recibirá en cada iteración un elemento de la colección sobre la que ejecutar el bloque de instrucciones.

```
PS C:\Users\adria> $letrasArray = "a","b","c","d","e","f"
PS C:\Users\adria> foreach ($letras in $letrasArray) {Write-Host $letras}
a
b
c
d
e
f
PS C:\Users\adria>
```

La otra manera es usando un *foreach-object*, que pasa la colección de objetos al *foreach* a través de un pipeline. Además, esta permite la ejecución de un bloque de script antes o después del procesamiento principal gracias a los parámetros *-begin* y *-end*.

4.2 Estructuras condicionales

Son **estructuras de programación que permiten tras evaluar una condición, decidir si se ejecuta un bloque de instrucciones** o se ejecuta otro bloque distinto o ninguno, es decir, permiten tomar decisiones a la hora de programar.

4.2.1 If, Else, Elseif

Esta estructura permite ejecutar un bloque si la condición del *if* es verdadera, mientras que si es falsa se ejecuta el bloque perteneciente al *else*. Con la instrucción *elseif* podemos hacer esta estructura más potente, ya que si la condición del *if* no se cumple permite evaluar las condiciones que aparezcan en distintos *elseif* antes de decidir ejecutar el bloque del *else*.

```
PS C:\Users\adria> $a = 1
PS C:\Users\adria> if ($a -eq 1){Write-Host "El valor es 1"} elseif ($a -eq 2){Write-Host "El valor es 2"} else {Write-Host "El valor no es ni 1 ni 2"}
El valor es 1
PS C:\Users\adria>
```

Condición *for* para **\$a** igual a 1, 2 o a ninguno no sea igual o mayor que 5

4.2.2 Switch

La instrucción *switch* permite **sustituir una serie de instrucción *if*, *elseif* y *else* por una sola instrucción *switch***.

```
PS C:\Users\adria> $a = 3
PS C:\Users\adria> switch ($a) { 1{"Uno"} 2{"Dos"} 3{"Tres"} 4{"Cuatro"} }
Tres
PS C:\Users\adria>
```

En ella se pasa una variable, y se comprueba con los valores de dentro del *switch*, y si el valor de la variable coincide con uno de los diferentes valores del *switch* se ejecutan las instrucciones correspondientes a ese valor. Además, se

puede añadir un valor **Default**, que se ejecutaría en caso de que no se produjera ninguna coincidencia anteriormente.

Switch puede recibir expresiones regulares si se especifica el parámetro -*regex*. Si hay varias coincidencias se ejecutarán todos los bloques que tuvieran coincidencia. Para evitar esto se emplea la palabra *break* que provoca la salida de la ejecución de la instrucción *switch*, haciendo que solo se produzca una coincidencia. Debido a esto es importante si se coloca la palabra clave *break* el orden de los valores a comparar de dentro de la instrucción *switch*.

5 Funciones

Si se suelen escribir scripts o comandos únicos de una línea de PowerShell y a menudo se tienen que modificar para distintos escenarios, es bastante probable que sean buenos candidatos para ser convertidos en **funciones**. Con el objetivo de ejecutar cierto tipo de información de forma eficiente y rápida.

Siempre es preferible escribir funciones ya que están más orientadas a las herramientas. A su vez, se pueden colocar esas funciones en un módulo de script, colocar ese módulo en **\$env:PSModulePath** y llamar a las funciones sin necesidad de buscar físicamente dónde se guardan.

5.1 Nomenclatura

Para poder nombrar a las funciones de PowerShell, se usa algún nombre en **Pascal Case** con un verbo aprobado y un sustantivo en singular. Es recomendable agregar un sufijo al nombre. En el mismo PowerShell hay una lista específica de verbos aprobados que se pueden obtener mediante la ejecución de **Get-Verb**

```
PowerShell
Get-Verb | Sort-Object -Property Verb
```

Output	
Verb	Group
----	-----
Add	Common
Approve	Lifecycle
Assert	Lifecycle
Backup	Data
Block	Security
Checkpoint	Data
Clear	Common
Close	Common
Compare	Data
Complete	Lifecycle
Compress	Data
Confirm	Lifecycle
Connect	Communications

Aquí se detallan algunos ejemplos del output del comando **Get-Verb**, ordenando por Verbo.

Es importante elegir un verbo aprobado en PowerShell al agregar funciones a un módulo. Si se elige un verbo **no aprobado**, el módulo generará un mensaje de advertencia en el momento de la carga. Ese mensaje de advertencia hará que las funciones no parezcan profesionales además de que este tipo de verbos limitan la detectabilidad de las funciones.

5.2 Funciones Simples

Para declarar este tipo de funciones se usa una palabra clave de función, seguida del nombre de dicha función y, a continuación, una llave de apertura y otra de cierre. El código que ejecutará la función se encuentra dentro.

```
PowerShell

function Get-Version {
    $PSVersionTable.PSVersion
}
```

Esta función simple devuelve la **versión** del PowerShell en dónde se esté ejecutando.

```
PowerShell

Get-Version

Output

Major  Minor  Build  Revision
-----
5      1      14393  693
```

Es bastante probable que surjan **conflictos** en los nombres de las funciones con un nombre como **Get-Version** y los comandos predeterminados del PowerShell o los comandos que otros usuarios puedan escribir. Esta es la razón por la que se recomienda prefijar los sustantivos.

Si se quieren ver las funciones que estén cargadas en memoria se usará **PSDrive**:

```
PowerShell

Get-ChildItem -Path Function:\Get-*Version

Output

CommandType  Name                Version  Source
-----
Function     Get-Version
Function     Get-PSVersion
Function     Get-MrPSVersion
```

Para poder eliminar las funciones de la sesión actual del PowerShell habría que de la propia función **PSDrive** o cerrar y volver a abrir. Con el *cmdlet* **Remove-Item** podremos cumplir el objetivo.

Además, es posible eliminar un módulo entero de la memoria de la sesión actual, pero hay que tener en cuenta que no se quitarán ni del sistema ni del disco.

PowerShell

```
Remove-Module -Name <ModuleName>
```

5.3 Parámetros

Lo más lógico y eficiente es no asignar valores de forma estática, si no usar parámetros y variables. Para asignar los nombres de parámetros se debe utilizar el mismo nombre que de los **cmdlets** predeterminados siempre que sea posible.

PowerShell

```
function Test-MrParameter {  
    param (  
        $ComputerName  
    )  
  
    Write-Output $ComputerName  
}
```

Se puede observar como el nombre del parámetro es **\$ComputerName**, de esta forma la función estará estandarizada como los cmdlets predeterminados.

A modo de aclaración, se podría crear una función que muestre todos los comandos del sistema y devolver el número de parámetros específicos que tienen cada uno de ellos.

PowerShell

```
function Get-MrParameterCount {  
    param (  
        [string[]]$ParameterName  
    )  
  
    foreach ($Parameter in $ParameterName) {  
        $Results = Get-Command -ParameterName $Parameter -ErrorAction SilentlyContinue  
  
        [pscustomobject]@{  
            ParameterName = $Parameter  
            NumberOfCmdlets = $Results.Count  
        }  
    }  
}
```

Si ejecutamos la función y le pasamos parámetros de tipo **String** para ver si es capaz de “matchear” con los parámetros de las funciones, obtendríamos la solución.

PowerShell

```
Get-MrParameterCount -ParameterName ComputerName, Computer, ServerName, Host, Machine
```

Output

ParameterName	NumberOfCmdlets
ComputerName	39
Computer	0
ServerName	0
Host	0
Machine	0

Es importante añadir que la instrucción `param` permite definir uno o varios parámetros. Las definiciones de parámetros están separadas por una coma (,).

5.4 Funciones Avanzadas

Este tipo de funciones se diferencian de las simples en que las avanzadas cuentan con una serie de parámetros comunes que se agregan a la función automáticamente. En estos parámetros se incluyen **Verbose** y **Debug**.

Para poder ver los parámetros comunes de una función, se hace uso del recurso `Get-Command`. Por ejemplo, la función `Test-MrParameter`, definida antes, no cuenta con parámetros comunes.

```
PowerShell

function Test-MrParameter {

    param (
        $ComputerName
    )

    Write-Output $ComputerName
}
```

Para poder cerciorar nuestra teoría se tendría que escribir `Get-Command -Name Test-MrParameter -Syntax`.

Estos nos devolvería la sintaxis de la función que a su vez mostraría los parámetros comunes que en este caso no existirían.

Si se quieren observar todos los parámetros utilizaríamos `Get-Command` de la siguiente manera: `(Get-Command -Name Test-MrParameter).Parameter.Keys`

Obteniendo como output el único parámetro que se le ha asignado a la función.

```
Output

ComputerName
```

Para poder convertir nuestra función en avanzada debemos de incluir `CmdletBinding`, que se encargará de agregar los parámetros automáticamente, es importante recalcar que `CmdletBinding` necesita un bloque `param` en su sintaxis, pero éste puede encontrarse vacío.

```
PowerShell

function Test-MrCmdletBinding {

    [CmdletBinding()] #<<-- This turns a regular function into an advanced function
    param (
        $ComputerName
    )

    Write-Output $ComputerName
}
```

Si ahora realizamos el comando `Get-Command -Name Test-MrParameter -Syntax` veremos:

```
Output
Test-MrCmdletBinding [[-ComputerName] <Object>] [<<CommonParameters>]
```

Y si usamos `(Get-Command -Name Test-MrParameter).Parameter.Keys` veremos los parámetros reales, incluidos los comunes:

```
Output
ComputerName
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
InformationVariable
OutVariable
OutBuffer
PipelineVariable
```

Validación de Parámetros

Se debe poder ejecutar el código siguiendo una entrada válida para permitir que éste cuente con una posible ruta de acceso, es decir, es necesario validar la entrada bien al principio.

Es importante cerciorarse de las variables que utilizan los parámetros especificando el tipo de datos entrantes.

```
PowerShell
function Test-MrParameterValidation {
    [CmdletBinding()]
    param (
        [string]$ComputerName
    )

    Write-Output $ComputerName
}
```

En este caso se observa que el parámetro **\$ComputerName** es de tipo **String**, pero solamente se asigna un valor único para el nombre de equipo. Si se da más de un nombre mediante una lista separada por comas se generará un error.

Sin embargo, se podría omitir el valor del parámetro **ComputerName** a la hora de llamar a la función, pero se requiere un valor para que la función se ejecute correctamente. Aquí surge la utilidad del parámetro **Mandatory** que, como su propio nombre indica vuelve obligatoria la asignación del valor al parámetro de la función.

```
PowerShell
function Test-MrParameterValidation {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [string]$ComputerName
    )

    Write-Output $ComputerName
}
```

Ahora si no se especifica ningún valor para el parámetro, la función solicitará uno en la gestión de errores.

Para poder permitir varios argumentos, se podría asignar una lista en vez de un único **String**, usando la sintaxis **String[]**, permitiendo una matriz de cadenas.

Una asignación válida y comprensible, sería la asignación de valores predeterminados en caso de que no se le asigne ninguno. El problema es que los valores predeterminados no se pueden usar con parámetros obligatorios. En su lugar, se deberá usar el atributo de validación de parámetros **ValidateNotNullOrEmpty** con un valor predeterminado.

```
PowerShell

function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    Write-Output $ComputerName
}
```

Es importante intentar no usar valores estáticos, en este caso, se ha utilizado **\$env:COMPUTERNAME** como valor predeterminado, que se traduce automáticamente en el nombre del equipo local si no se nos proporciona un valor.

Resultado Detallado

Aunque los comentarios son muy útiles, en términos de programación, los usuarios finales no los verán nunca, a no ser que examinen el propio código. Aquí existe un ejemplo de comentarios únicamente legibles por aquellas personas que vean el código directamente.

```
PowerShell

function Test-MrVerboseOutput {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    foreach ($Computer in $ComputerName) {
        #Attempting to perform some action on $Computer <-- Don't use
        #inline comments like this, use write verbose instead.
        Write-Output $Computer
    }
}
```

Si se requiere que la función exponga cierto tipo de información en formatos opcionales, se deberá usar **Write-Verbose** en lugar de comentarios insertados.

```

PowerShell

function Test-MrVerboseOutput {
    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    foreach ($Computer in $ComputerName) {
        Write-Verbose -Message "Attempting to perform some action on $Computer"
        Write-Output $Computer
    }
}

```

Esto provocará que cuando se llame a la función sin colocar el parámetro **Verbose**, no se mostrará el resultado detallado, pero si se utiliza expondrá lo escrito anteriormente en la ejecución de la función.

Entrada de Canalización

Si se requiere que la función acepte entradas de canalización, se necesitará cierta codificación adicional. Es importante resaltar que los comandos pueden aceptar la entrada de canalización por valor o por nombre de propiedad. Puede escribir las funciones de la misma manera que los comandos nativos, para que acepten uno o ambos tipos de entrada.

Para aceptar la entrada de canalización por valor, se ha de especificar el atributo de parámetro *ValueFromPipeline* para ese parámetro concreto. Hay que tener en cuenta que sólo puede aceptar la entrada de canalización por valor de uno de cada tipo de datos. Por ejemplo, si se tienen dos parámetros que aceptan la entrada de cadena, sólo uno de ellos puede aceptar la entrada de canalización por valor, porque, si lo especificó para ambos parámetros de la cadena, la entrada de canalización no sabrá con cuál se debe enlazar. Este es otro motivo por el que se llama a este tipo de entrada de canalización por tipo, en lugar de por valor.

La entrada de canalización se incluye en un elemento cada vez, de manera similar a la forma en la que se administran los elementos en un bucle "foreach". Como mínimo, se requiere un bloque 'process' para procesar cada uno de los elementos si se va a aceptar una matriz como entrada. Si sólo va a aceptar o valor único como entrada, no es necesario un bloque process, pero sigue siendo recomendable especificarlo para que sea coherente.

```

PowerShell

function Test-MrPipelineInput {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
            ValueFromPipeline)]
        [string[]]$ComputerName
    )

    PROCESS {
        Write-Output $ComputerName
    }
}

```

Aceptar la entrada de canalización por nombre de propiedad es similar, salvo que se especifica con *ValueFromPipelineByPropertyName* y se puede especificar para cualquier número de parámetros, independientemente del tipo de datos. La clave es que la salida del comando que se está canalizando debe tener un nombre que coincida con el nombre del parámetro o un alias de parámetro de la función.

```
PowerShell

function Test-MrPipelineInput {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                    ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        Write-Output $ComputerName
    }
}
```

Además, se pueden usar de forma opcional los bloques **BEGIN** y **END**. El primero se usaría antes del bloque **PROCESS** para realizar cualquier trabajo inicial antes de recibir los elementos de canalización. Los valores que se canalizan no son accesibles en el bloque **BEGIN**. El bloque **END** se especificaría después del bloque **PROCESS** y se utilizaría para la limpieza una vez que se hubieran procesado todos los elementos canalizados.

5.5 Tratamiento de Errores

Como en muchos otros lenguajes, es posible generar excepciones a diversos parámetros que se le asignen o a funcionalidades del código.

La forma más moderna de controlar los errores es mediante la famosa sentencia *Try/Catch*.

```
PowerShell

function Test-MrErrorHandling {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                    ValueFromPipeline,
                    ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Test-WSMan -ComputerName $Computer
            }
            catch {
                Write-Warning -Message "Unable to connect to Computer: $Computer"
            }
        }
    }
}
```

Aunque la función que se muestra en el ejemplo anterior usa el control de errores, también genera una excepción no controlada porque el comando no genera un error de terminación. Sólo se detectan los errores de terminación. Para poder convertir un error de no terminación en uno de terminación se debe especificar el parámetro **ErrorAction** o **Stop**.

```

PowerShell

function Test-MrErrorHandling {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                    ValueFromPipeline,
                    ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Test-WSMan -ComputerName $Computer -ErrorAction Stop
            }
            catch {
                Write-Warning -Message "Unable to connect to Computer: $Computer"
            }
        }
    }
}

```

Es aconsejable no modificar la variable **\$ErrorActionPreference** a menos que sea absolutamente necesario. Si se usa algo parecido a .NET directamente desde la función de PowerShell, no puede especificar **ErrorAction** en el propio comando. En ese caso, es posible tener que modificar la variable global, aunque después de probar el comando es importante volver a cambiarla.

6 Cmdlets

Un cmdlet es un comando ligero que se usa en el entorno de PowerShell. El tiempo de ejecución de PowerShell invoca estos cmdlets en el contexto de los scripts de automatización que se proporcionan en la línea de comandos.

Éstos realizan una acción y normalmente devuelven un objeto Microsoft .NET al siguiente comando de canalización.

Los siguientes términos se usan con frecuencia en la documentación del cmdlet de PowerShell:

- **Atributo:** se usa para declarar una clase de cmdlet como un cmdlet. Aunque PowerShell usa otros opcionales el cmdlet es obligatorio.
- **Parámetro:** propiedades públicas que definen los parámetros que están disponibles para el usuario o la aplicación que ejecuta el cmdlet.
- **Conjunto de parámetros:** grupo de parámetros que pueden usarse en el mismo comando para realizar una acción específica.
- **Parámetro dinámico:** parámetro que se agrega al cmdlet en tiempo de ejecución.
- **Transacción:** grupo lógico de comandos que se tratan como una sola tarea. La tarea genera un error automáticamente si se produce uno en algún comando del grupo y el usuario tiene la opción de aceptar o rechazar las acciones realizadas dentro de la transacción.

Los cmdlets se diferencian de los comandos en distintos aspectos:

- Los cmdlets son instancias de .NET; no son ejecutables independientes.
- Los cmdlets se pueden crear desde pocas líneas de código.

- Los cmdlets no realizan su propio análisis, presentación de errores o formato de salida. Es el tiempo de ejecución de PowerShell quien lo administra.
- Los cmdlets procesan objetos de entrada de canalización en lugar de flujos de texto y suelen ofrecer objetos como salida de canalización.
- Los cmdlets están orientados a registros porque se procesan un solo objeto cada vez.

PowerShell define varios atributos de .NET que se usan para administrar cmdlets y para especificar la funcionalidad común proporcionada por PowerShell y que el cmdlet puede necesitar. Por ejemplo, los atributos se usan para designar una clase como un cmdlet, especificar los parámetros del cmdlet y solicitar la validación de la entrada que los desarrolladores de cmdlets no tengan que implementar esa funcionalidad en su código de cmdlet.

Aquí se encuentra un ejemplo sencillo de un cmdlet:

```
using System.Management.Automation; // Windows PowerShell assembly.

namespace SendGreeting
{
    // Declare the class as a cmdlet and specify the
    // appropriate verb and noun for the cmdlet name.
    [Cmdlet(VerbsCommunications.Send, "Greeting")]
    public class SendGreetingCommand : Cmdlet
    {
        // 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 user name and write out a
        // greeting to the user by calling the WriteObject
        // method.
        protected override void ProcessRecord()
        {
            WriteObject("Hello " + name + "!");
        }
    }
}
```

Los pasos para escribir un cmdlet son:

1. Declarar la clase como un cmdlet, usando el atributo `Cmdlet`.
2. Especificar el nombre de la clase.
3. Especificar que el cmdlet se derive de cualquiera de las clases:
 - a. `System.Management.Automation.cmdlet`.
 - b. `System.Management.Automation.PSCmdlet`
4. Definir los parámetros con `Parameter`.
5. Invalidar el método de procesamiento de entrada que procesa la entrada.

Para nombrar los cmdlets se usa un verbo y un sustantivo, por ejemplo, `Get-Command` que obtiene información de todos los cmdlets registrados en el Shell

de comandos. El verbo identifica la acción que realiza el cmdlet y el nombre identifica el recurso en el que el cmdlet realiza su acción.

7 Objetos

PowerShell es un lenguaje y shell orientado a objetos. Esta es una desviación de las conchas tradicionales como cmd y Bash.

Estos shells tradicionales se enfocan en texto, también conocido como cadenas, y aunque siguen siendo útiles, tienen capacidades limitadas. Casi todo en PowerShell es un objeto.

Los objetos tienen muchos tipos de información diferente asociada. En PowerShell, esta información a veces se denomina "miembros". Un miembro de objeto es un término genérico que se refiere a toda la información asociada con un objeto (equivalencia a atributos y métodos en programación orientada a objetos). Para descubrir información sobre un objeto (miembros), puede usar el cmdlet `Get-Member`. (Un cmdlet es un script ligero de Windows Powershell que realiza una sola función. Un comando, en este contexto, es un pedido específico de un usuario al sistema operativo del ordenador o a una aplicación para realizar un servicio, como "Mostrar todos mis archivos" o "Ejecutar un programa". Se expresa como un par verbo-sustantivo. Además, cada cmdlet tiene un archivo de ayuda al que se puede acceder escribiendo '`Get-help <cmdlet-name> -Detailed`' o '`help <cmdlet-name> -Full`'). El cmdlet `Get-Member` nos permite encontrar propiedades, métodos, etc. disponibles para cualquier objeto en PowerShell.

Por ejemplo, la invocación del comando `Get-Service -ServiceName w32time`, nos devuelve información parcial del objeto `w32time` el estado del servicio con su nombre. Como la devolución de la llamada a este comando es un objeto, podemos redirigir la salida al comando `Get-Member` para así ver toda la información (miembros) de dicho objeto. Ej: `Get-Service -Name w32time | Get-Member`

```

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

TypeName: System.ServiceProcess.ServiceController

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

```

Cada objeto tiene un esquema. El "esquema" de un objeto es una especie de plantilla que contiene el plano para crear un objeto. Ese plano se llama tipo. Cada objeto en PowerShell tiene un tipo específico. Cada objeto tiene un plano a partir del cual fue creado. Un tipo de objeto está definido por una clase. Los objetos son instancias de clases con un tipo particular. **TypeName** dice qué tipo de objeto se devolvió. En este ejemplo, se devolvió un objeto **System.ServiceProcess.ServiceController**.

Uno de los conceptos más importante sobre los objetos que se debe comprender son las propiedades. Las propiedades son atributos que describen un objeto. Un objeto puede tener muchas propiedades diferentes adjuntas que representan varios atributos. Una de las formas más sencillas de descubrir qué propiedades existen en los objetos es mediante el cmdlet **Get-Member** y usar el parámetro **MemberType**. Ej: **Get-Service -Name w32time | Get-Member -MemberType Property**.

```
PS C:\Users\lukas> Get-Service -Name w32time | Get-Member -MemberType Property

    TypeName: System.ServiceProcess.ServiceController

Name            MemberType Definition
-----
CanPauseAndContinue Property bool CanPauseAndContinue {get;}
CanShutdown      Property bool CanShutdown {get;}
CanStop          Property bool CanStop {get;}
Container        Property System.ComponentModel.IContainer Container {get;}
DependentServices Property System.ServiceProcess.ServiceController[] DependentServices {get;}
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;}
Status           Property System.ServiceProcess.ServiceControllerStatus Status {get;}

```

De esta forma podemos obtener cualquier miembro de un objeto, pero si deseásemos ver el valor concreto de un miembro en lugar de la información de su esquema podríamos ejecutar el siguiente comando: **Get-Service -Name w32time | Select-Object -Property 'Status'**. Podemos tener aún más control sobre la salida utilizando Sort-Object para ordenar la salida según los valores de la propiedad/es que deseemos. También podríamos utilizar Where-Object, seleccionando así solo objetos según los valores que se especifiquen.

```
PS C:\Users\lukas> Get-Service -Name w32time | Select-Object -Property 'Status'

Status
-----
Stopped

```

Existe un **MemberType** que son los alias. Los alias son seudónimos de nombres de propiedades. A veces pueden dar a las propiedades un nombre más intuitivo. Cuando una propiedad tiene un alias, puede hacer referencia al valor de esa propiedad utilizando el nombre del alias en lugar del nombre real de la propiedad.

```
PS C:\Users\lukas> Get-Service -Name w32time | Get-Member -MemberType 'AliasProperty'

    TypeName: System.ServiceProcess.ServiceController

Name            MemberType Definition
-----
Name            AliasProperty Name = ServiceName
RequiredServices AliasProperty RequiredServices = ServicesDependedOn

```

Los métodos son las acciones que se pueden realizar en un objeto. Al igual que las propiedades, puede descubrir métodos en un objeto mediante el cmdlet **Get-Member**. Ej: **Get-Service -Name w32time | Get-Member -MemberType Method**.

```
PS C:\Users\lukas> Get-Service -Name w32time | Get-Member -MemberType Method_

TypeName: System.ServiceProcess.ServiceController

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

Los alias, métodos y propiedades son los miembros más comunes, pero existen también otros tipos

- Script Property: se utilizan para calcular los valores de propiedad.
- Note Property: se utilizan para nombres de propiedades estáticas.
- Property Sets: son como alias que contienen exactamente lo que implica el nombre; conjuntos de propiedades. Por ejemplo, ha creado una propiedad personalizada llamada Specs para su función **Get-CompInfo**. **Specs** es en realidad un subconjunto de las propiedades Cpu, Mem, Hdd, IP. El propósito principal de los conjuntos de propiedades es proporcionar un nombre de propiedad único para concatenar un grupo de propiedades.

7.1 Pipelines

Como hemos visto en muchos de los ejemplos mostrados hasta ahora, muchas veces la salida de un comando se puede usar como entrada para otro comando. Dependiendo de cuán completa sea la ayuda de los comandos, puede incluir una sección de ENTRADAS y SALIDAS. Ej: **help Stop-Service -Full**.

```
ENTRADAS
System.String[]
System.ServiceProcess.ServiceController[]

SALIDAS
System.ServiceProcess.ServiceController
```

La sección ENTRADAS indica que un objeto `ServiceController` o `String` se puede “canalizar” al cmdlet `Stop-Service`. No le dice qué parámetros aceptan ese tipo de entrada.

Una de las formas más sencillas de determinar esa información es examinar los diferentes parámetros en la versión completa de la ayuda del cmdlet `Stop-Service: help Stop-Service-Full`.

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

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

-InputObject <ServiceController[]>
    Specifies ServiceController objects that represent the services to stop.
    You can type the name of the variable that contains the objects, or type a command or expression
    that produces the objects.

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

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

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

En esta salida puede verse que el parámetro `DisplayName` no acepta entrada de canalización, el parámetro `InputObject` acepta entrada de canalización por valor para objetos `ServiceController` y el parámetro `Name` acepta entrada de canalización por valor para objetos de cadena. También acepta entrada de canalización por nombre de propiedad.

Cuando un parámetro acepta entrada de canalización tanto por nombre de propiedad como por valor, siempre lo intenta primero por valor. Si el valor o tipo falla, lo intenta por el nombre de la propiedad. Esto significa que si canalizamos los resultados de un comando que produce un tipo de objeto `ServiceController` a `Stop-Service`, vincula esa entrada al parámetro `InputObject`. Pero si canaliza los resultados de un comando que produce una salida de cadena a `Stop-Service`, lo vincula al parámetro `Name`.

Si canaliza los resultados de un comando que no produce un objeto `ServiceController` o `String` a `Stop-Service`, pero produce una salida que

contiene una propiedad llamada **Name**, entonces vincula la propiedad **Name** de la salida al parámetro **Name** de **Stop-Service**.

La ejecución del comando `Get-Service -Name w32time | Get-Member`, nos indica que **Get-Service** nos devuelve un **ServiceController**, y el cmdlet **Stop-Service** admite como entrada un **String** o un **ServiceController**, por lo que crear un pipeline sería posible.

Esto hace de los pipelines un concepto muy poderoso. Ya que, a diferencia de otros shells, no existe una limitación a pasar solo cadenas, podemos pasar también objetos al siguiente comando. Cuando se trabaja con la canalización de PowerShell, hay una variable automática que es de particular importancia, **\$PSItem**. **\$PSItem** contiene el objeto actual en la canalización. Por ejemplo, `Get-Process | ForEach-Object {$PSItem}`.

8 Módulos

Una muy interesante novedad que se incorpora ya desde versiones muy prematuras, como la V2.0 del Powershell, son los **módulos**. Los módulos permiten a los administradores, desarrolladores, etc. dividir y organizar el código en unidades propias, que se reutilizables y compartibles con facilidad. Haciendo una analogía inicial darle forma en la mente a la propuesta, con la mayoría de lenguajes de programación orientada a objetos, sería como una especie de **paquete**: se ejecuta en un contexto que está contenido en el propio módulo, es decir: permite definir variables, **alias**, **funciones**, etc., de forma privada o pública (que se puedan invocar sólo desde el propio módulo (**privadas**), o directamente desde la línea de comandos o un script de Powershell (**públicas**) una vez se ha importado el módulo. Esto significa que podemos crear un script y que éste se ejecute en un espacio de trabajo restringido.

La ventaja de los módulos es que son fácilmente transportables y fácil de compartir para que otros usuarios puedan emplearlos, basta con hacerse con el archivo para disfrutar del módulo.

8.1 Componentes y tipos de módulo

Un módulo se puede componer de cuatro componentes básicos:

1. Algún tipo de archivo de **código**, normalmente un script de PowerShell o un ensamblado de *cmdlet* administrado.
2. Cualquier cosa que pueda necesitar el archivo de código anterior, como ensamblados adicionales, archivos de ayuda, scripts, etc.
3. Un archivo de **manifiesto** que describe los archivos anteriores, así como almacenar metadatos.

4. **Directorio que contiene todo** el contenido anterior y se encuentra donde PowerShell puede encontrarlo de forma razonable.

Este equipo > Windows (C:) > Windows > System32 > WindowsPowerShell > v1.0 > Modules > Dism				
Nombre	Fecha de modificación	Tipo	Tamaño	
en	13/03/2021 20:51	Carpeta de archivos		
es	07/12/2019 15:55	Carpeta de archivos		
Dism.Format.ps1xml	07/12/2019 10:09	Documento XML ...	26 KB	
Dism.psd1	07/12/2019 10:09	Archivo de datos ...	3 KB	
Dism.psm1	07/12/2019 10:09	Módulo de script ...	2 KB	
Dism.Types.ps1xml	07/12/2019 10:09	Documento XML ...	20 KB	
Microsoft.Dism.PowerShell.dll	07/12/2019 10:09	Extensión de la ap...	130 KB	

Ejemplo de un módulo completo (manifiesto, script y documentación)

Ninguno de estos componentes es por si solo realmente imprescindible. Por ejemplo, un módulo puede ser técnicamente solo un script almacenado en un archivo. psm1. También puede tener un módulo solo de manifiesto, que se usa principalmente para fines organizativos. Combinando los **cuatro** componentes mencionados podemos definir los diferentes tipos de módulos.

8.1.1 Módulos de script (Script)

```
function Show-Calendar {  
    param(  
        [DateTime] $start = [DateTime]::Today,  
        [DateTime] $end = $start,  
        $firstDayOfWeek,  
        [int[]] $highlightDay,  
        [string[]] $highlightDate = [DateTime]::Today.ToString()  
    )  
  
    #actual code for the function goes here see the end of the topic for the complete code sample  
}
```

Un módulo de script puede ser tan simple como una sola función

Un *módulo de script* es un **archivo (. psm1)** que contiene cualquier código válido de Windows PowerShell, de tipo script. Los desarrolladores y administradores de scripts pueden usar este tipo de módulo para crear módulos cuyos miembros incluyen funciones, variables, etc. En esencia, el módulo más habitual que más se ajusta a la definición anterior de módulo.

Puede incluir (o no) un archivo de manifiesto para incluir otros recursos en el módulo, como archivos de datos, otros módulos dependientes o scripts en tiempo de ejecución. Los archivos de manifiesto también son útiles para realizar el seguimiento de metadatos, como la creación y la información de control de versiones.

Por último, un módulo de script, como cualquier otro módulo debe guardarse en una carpeta que PowerShell pueda detectar de forma razonable.

8.1.2 Módulos binarios (Binary)

```
C# Copiar

using System.Management.Automation; // Windows PowerShell namespace.

namespace ModuleCmdlets
{
    [Cmdlet(VerbsDiagnostic.Test, "BinaryModuleCmdlet1")]
    public class TestBinaryModuleCmdlet1Command : Cmdlet
    {
        protected override void BeginProcessing()
        {
            WriteObject("BinaryModuleCmdlet1 exported by the ModuleCmdlets module.");
        }
    }

    [Cmdlet(VerbsDiagnostic.Test, "BinaryModuleCmdlet2")]
    public class TestBinaryModuleCmdlet2Command : Cmdlet
    {
        protected override void BeginProcessing()
        {
            WriteObject("BinaryModuleCmdlet2 exported by the ModuleCmdlets module.");
        }
    }

    [Cmdlet(VerbsDiagnostic.Test, "BinaryModuleCmdlet3")]
    public class TestBinaryModuleCmdlet3Command : Cmdlet
    {
        protected override void BeginProcessing()
        {
            WriteObject("BinaryModuleCmdlet3 exported by the ModuleCmdlets module.");
        }
    }
}
```

Un *módulo binario* es un **ensamblado de .NET Framework (.dll)** que contiene **código compilado**, como C#. Los desarrolladores de cmdlets pueden usar este tipo de módulo para compartir cmdlets, proveedores y mucho más. (Los complementos existentes también se pueden usar como módulos binarios). En comparación con un módulo de script, un módulo binario le permite crear cmdlets que son más rápidos o usan características (como multithreading) que no son tan fáciles de codificar en scripts de Windows PowerShell.

Al igual que con los módulos de script, puede incluir un archivo de manifiesto para describir los recursos adicionales que usa el módulo y para realizar un seguimiento de los metadatos sobre el módulo.

8.1.3 Módulos dinámicos (Dynamic)

```
PS C:\Users\diegu> New-Module -ScriptBlock {function Hola($name) {"Hola, $name!"}
>> function Adios($name) {"Adiós, $name!"}
>> function Cumple($yold, $name) {"Feliz $yold º cumpleaños, $name"}} -name ModuloFelicizador | Import-Module | Get-Module

PS C:\Users\diegu> Get-Module

ModuleType Version      Name                               ExportedCommands
-----
Script      0.0             GreetingModule                   Hello
Manifest    3.1.0.0         Microsoft.PowerShell.Utility     {Add-Member, Add-Type, Clear-Variable, Compare-Object...}
Script      0.0             ModuloFelicizador                {Adios, Cumple, Hola}
Script      2.0.0           PSReadline                      {Get-PSReadLineKeyHandler, Get-PSReadLineOption, Remove-PSReadLineKeyHandler, Set-PSReadLineKeyHandler...}

PS C:\Users\diegu> Cumple("20")("Diego")
Feliz 20 º cumpleaños, Diego
PS C:\Users\diegu> _
```

Ejemplo de creación de módulo de script dinámico

Un *módulo dinámico* es aquel que no se encuentra almacenado en una carpeta, sino creado en tiempo de ejecución y que se mantiene hasta que se cierra la sesión de PowerShell. Es un tipo especial de módulo no persistente.

8.1.4 Módulos del manifiesto (Manifest)

Manifiestos

Es ese fichero de datos de PowerShell (.psd1), que nos permite:

- Definir prerequisites: otros módulos, scripts, etc., que deben ser cargados antes que el propio módulo, por necesitarlo para su funcionamiento
- Describir los contenidos y atributos del módulo.
- Limitar qué es exportado por los módulos, ensamblados, etc.
- Determinar cómo se procesan los componentes.

Los manifiestos **no son requeridos** por los módulos.

Los manifiestos pueden referenciar scripts (*.ps1), ficheros de módulos (*.psm1), otros ficheros de manifiesto (*.psd1), ficheros de formatos y tipos (*.ps1xml), ensamblados con proveedores y Cmdlets (*.dll), etc. El uso de un manifiesto en la carpeta de un módulo permite poder referenciar todos estos ficheros como una simple unidad.

Los principales tipos de información de un manifiesto son:

- Metadatos del módulo, como versión, autor, descripción, etc.
- Prerequisites necesarios para importar el módulo, como la versión de Windows PowerShell, la versión de Common Language Runtime (CLR) y los módulos requeridos.
- Directivas de procesamiento referentes a scripts, formatos y tipos a procesar.

Algunas de las palabras clave de los manifiestos son:

- **ModuleToProcess:** el módulo de script o ensamblado con el que está asociado el manifiesto. También hace referencia al módulo raíz cuando se trata de módulos anidados. Cuando no se especifica ningún módulo el propio fichero de manifiesto se convierte en el módulo raíz y se hace referencia a él como módulo de manifiesto.
- **ModuleList:** Lista todos los módulos integrantes del módulo.
- **Version:** número de versión del módulo.
- **GUID:** identificador del módulo.
- **Author:** autor del módulo.
- **CompanyName:** nombre de la empresa o vendedor del módulo.
- **Description:** descripción del módulo.
- **PowerShellVersion:** versión mínima de Windows PowerShell que puede utilizar el módulo. Los valores posibles, de momento, son 1.0 y 2.0.
- **RequiredModules:** módulos adicionales requeridos por el módulo del manifiesto. Windows PowerShell no importará los módulos de forma automática, solamente verifica que estén presentes. No obstante, sí pueden ser cargados por algún script incluido en el módulo. No tiene por qué definirse este parámetro.
- **RequiredAssemblies:** ensamblados (.dll) requeridos por el módulo. No tiene por qué definirse este parámetro.
- **ScriptsToProcess:** scripts que deben ser ejecutados cuando es importado el módulo. Permiten preparar el entorno de la misma manera que se usan scripts de logon. No tiene por qué definirse este parámetro.

- **TypesToProcess:** ficheros de tipos (.pslxml) asociados con el módulo. Consisten en tipo de .NET Framework usados por los componentes del módulo. Al importar el módulo, Windows PowerShell ejecuta **Update-TypeData** con los ficheros especificados. No tiene porqué definirse este parámetro.
- **FormatsToProcess:** ficheros de formato (.pslxml) asociados con el módulo. Contienen información de presentación de tipos .NET usados por los componentes del módulo. No tiene porqué definirse este parámetro.
- **NestedModules:** ficheros de módulo, scripts (.psml) y/o binarios (.dll) que son importados en la sesión propia del módulo, pero no en la sesión global. Esto significa que sus contenidos solo son visibles para el módulo, pero no para el usuario, a no ser que se use el parámetro Global al usar **Import-Module** para cargar el módulo o se exporten explícitamente desde el módulo raíz. Los módulos anidados se cargan en el orden en que están listados, lo cual es necesario tenerlo en cuenta si hay dependencias entre ellos. No tiene porqué definirse este parámetro.
- **FunctionsToExport:** restringe las funciones que son exportadas por el módulo. Esto permite que a pesar de que un módulo de tipo script (.psml) exporte determinadas funciones, desde el manifiesto se permita exportar solo unas determinadas, lo cual es interesante cuando hay módulos anidados. No tiene porqué definirse este parámetro.
- **CmdletsToExport:** lo mismo que **FunctionsToExport**, pero en referencia a Cmdlets.
- **VariablesToExport:** lo mismo que **FunctionsToExport**, pero en referencia a las variables
- **PrivateData:** datos pasados al módulo cuando es importado. Estos datos están disponibles utilizando la variable **\$args** dentro de los módulos de script. No es necesario especificar este parámetro.

El procedimiento recomendado para crear un manifiesto de módulo es usar el cmdlet **New-ModuleManifest**. Puede usar parámetros para especificar uno o más valores y claves predeterminados del manifiesto. El único requisito es asignar un nombre al archivo. **New-ModuleManifest** crea un manifiesto de módulo con los valores especificados e incluye las claves restantes y sus valores predeterminados. Si tiene que crear varios módulos, use **New-ModuleManifest** para crear una plantilla de manifiesto de módulo que se pueda modificar para los distintos módulos.

Un *módulo de manifiesto* es, por tanto, un módulo que utiliza un archivo de manifiesto para describir todos sus componentes, pero no tiene ningún tipo de ensamblado o script básico.

Puede usar las otras características de un módulo, como la capacidad de cargar ensamblados dependientes o ejecutar automáticamente ciertas secuencias de comandos previas al procesamiento. Es decir, es útil usar un módulo de manifiesto como una manera cómoda de **organizar recursos** que otros módulos van a usar, como los módulos anidados, los ensamblados, los tipos o los formatos.

8.2 Añadir un módulo

En WindowsServer. Cada rol instalado da también derecho a un módulo que se instala automáticamente. Para recibir roles de terceros debemos añadirlos a la carpeta **Modules**. Los módulos se presentan bajo la forma de carpetas que contienen uno o varios archivos. Estas carpetas se encuentran en las siguientes rutas:

RUTA	Información
%UserProfile%\Documents\WindowsPowerShell\Modules	Esta ubicación contiene los módulos del usuario en curso.

C:\Program Files\WindowsPowerShell\Modules	Ubicación común a todos los usuarios de la máquina.
%Windir%\System32\	Esta ubicación contiene todos los módulos proporcionados por Microsoft. Se comparten entre todos los usuarios de una máquina.

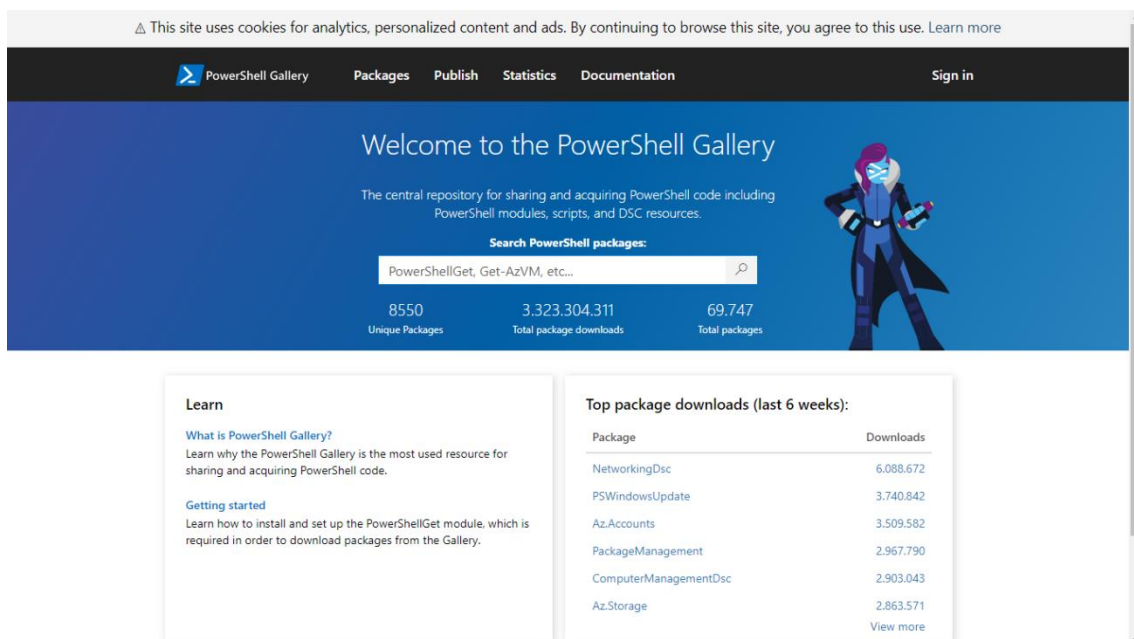
Tenga precaución, pues las ubicaciones para almacenar los módulos difieren entre Windows PowerShell y PowerShell Core. Sea la que sea, está referenciada en la variable de entorno **PSModulePath**.

Si añadimos una carpeta a la variable de entorno **PSModulePath**, PowerShell la analizará para determinar la presencia de módulos. Esto da flexibilidad para la organización de los módulos.

Cabe destacar de nuevo la versatilidad de los módulos y su facilidad de obtención de cualquier parte de la red:

Existen múltiples sitios desde los que descargar módulos:

- Repositorios públicos de módulos como PowerShell Gallery, que gestiona la propia compañía Microsoft.
- Empresas del sector que deciden integrar PowerShell para dar soporte a sus productos
- Pero también cualquier repositorio abierto al mundo o blogs relacionados con la administración de sistemas.



Portal de PowerShell Gallery – www.powershellgallery.com

8.3 Importar un módulo

Lo primero que debemos hacer si queremos trabajar con módulos ya escritos es importarlos. Una vez importados podremos utilizar los comandos que éstos contienen de forma totalmente transparente. Podemos importar módulos:

- De forma automática (a partir de PowerShell versión 3.0) ubicándolos en una carpeta que forme parte de nuestro path de módulos. Al hacerlo de este modo el módulo se importará la primera vez que ejecutemos un comando del mismo en la sesión actual. *Podemos desactivar este comportamiento modificando la variable \$PSModuleAutoloadingPreference (None).*
- Utilizando el cmdlet **Import-Module** pasando como parámetro la ruta de nuestro módulo de PowerShell.

8.4 Eliminar un módulo

Podemos eliminar los elementos de un módulo, como cmdlets y functions, de la sesión actual con **Remove-Module**. Este cmdlet no desinstala el módulo ni lo elimina de la máquina. Afecta solo a la sesión actual de PowerShell.

Si realmente los queremos eliminar del sistema bastará con quitarlos del path y el PowerShell dejará de tenerlos en cuenta.

Principales *cmdlets* para trabajar con módulos:

Cmdlet	Descripción
New-Module	Crea un nuevo módulo dinámico.
New-ModuleManifest	Crea un nuevo fichero de manifiesto (*.psd1), agrega sus valores y lo salva en una ruta especificada.
Import-Module	Agrega uno o más módulos a la sesión de PowerShell.
Get-Module	Especifica los elementos de un módulo que son exportados desde este. Cmdlets, funciones, variable y alias.
Export-ModuleMember [[-Function <String[]>] [-Cmdlet <String[]>] [-Variable <String[]>] [-Alias <String[]>] [<CommonParameters>]	Especifica los miembros del módulo que son exportados desde un módulo Cmdlets, funciones, variable y alias.
Remove-Module	Elimina los elementos de un módulo, como cmdlets y functions, de la sesión actual. Este cmdlet no desinstala el módulo ni lo elimina de la computadora. Afecta solo a la sesión actual de PowerShell.
Test-ModuleManifest	Verifica que los archivos que se enumeran en el archivo de manifiesto del módulo (.psd1) se encuentran realmente en las rutas especificadas. Está diseñado para ayudar a los autores de módulos a probar sus archivos de manifiesto.

Por otra parte, podemos obtener algunas funciones adicionales relacionadas con los módulos.

```
PS C:\Users\diegu> Get-Command -Name "*Module*" -CommandType "Function"

CommandType      Name                                Version      Source
-----
Function         Find-Module                        1.0.0.1      PowerShellGet
Function         Get-InstalledModule                1.0.0.1      PowerShellGet
Function         ImportSystemModules
Function         InModuleScope                     3.4.0        Pester
Function         Install-Module                    1.0.0.1      PowerShellGet
Function         Publish-Module                    1.0.0.1      PowerShellGet
Function         Save-Module                       1.0.0.1      PowerShellGet
Function         Uninstall-Module                  1.0.0.1      PowerShellGet
Function         Update-Module                     1.0.0.1      PowerShellGet
Function         Update-ModuleManifest             1.0.0.1      PowerShellGet
```

Variables referentes a los módulos

\$PSScriptRoot: Contiene el directorio desde el que el módulo es ejecutado. Permite que los scripts accedan a otros recursos. Por defecto señala al actual.

\$PSModulePath: Contiene una lista de localizaciones de módulos.

```
PS C:\Users\diegu> $Env:PSScriptRoot
PS C:\Users\diegu> $Env:PSModulePath
C:\Users\diegu\Documents\WindowsPowerShell\Modules;C:\Program Files\WindowsPowerShell\Modules;C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
PS C:\Users\diegu>
```

Salida de las variables de entorno *PSScriptRoot* y *PSModulePath*

9 Interacción con Windows Management Instrumentation (WMI)

9.1 Introducción

WMI es la implementación del estándar **Web-Based Enterprise Management**. Éste, es un conjunto de tecnologías de gestión que se basa en los estándares abiertos de Internet cuyo objetivo es unificar la gestión de plataformas y tecnologías dispares evolucionadas en un medio distribuido. La implementación de WMI se ha realizado en todos los sistemas operativos de Microsoft desde **Windows NT 4.0**. Funciona como una base de datos ofreciendo una amplia gama de información útil para monitorizar cualquier tipo de sistema basado en Windows. Como ya hemos comentado, viene instalado por defecto en todos los sistemas operativos antes mencionado, sin embargo, hay que cerciorarse de que esté correctamente activado. Como todo, se va actualizando y mejorando. Por ello cada versión de Windows implementa mejoras a WMI, nuevas clases WMI y nuevas funcionalidades para las clases WMI ya existentes.

Existen distintos tipos de categorías de tareas que se pueden realizar con WMI. Estos son algunos:

<u>Cuentas y dominios</u>	Obtener información como el dominio del equipo o el usuario que ha iniciado sesión actualmente
Hardware del equipo	Obtenga información sobre la presencia, el estado o las propiedades de los componentes de hardware
Software del Equipo	Obtenga información como qué software instala el Windows Installer (MSI) y las versiones de software.

Conexión al servicio WMI	Para obtener datos de WMI.
Fechas y Horas	Analizar o convertir el formato de fecha y hora.
Administración de Escritorio	Obtener datos o controlar los escritorios remotos.
Redes	Administrar y obtener información acerca de las conexiones y las direcciones IP o MAC.
Sistemas Operativos	Obtenga información sobre el sistema operativo, como la versión, si está activada o qué revisiones están instaladas.

9.2 Obtener un objeto de WMI

La tecnología WMI se utiliza principalmente para la administración del sistema de Windows. Esto se debe a la amplia gama de información que ofrece de manera uniforme. El cmdlet utilizado para obtener y así acceder a objetos WMI es **Get-CimInstance**.

9.3 Enumerar clases WMI

La mayor problemática de cara a los usuarios es que intentan descubrir qué pueden hacer con WMI. Para esto están las clases WMI. Estas describen los recursos que se pueden administrar. Cada clase consta con una serie de propiedades. Cabe destacar que existen cientos de clases WMI por lo que no procede poner todas aquí.

Con **Get-CimClass** podemos abordar este problema y obtener una lista de las clases WMI disponibles en el equipo local.

Para ello, escribimos lo siguiente en la consola de comandos:

```
Get-CimClass -Namespace root/CIMV2 |
  Where-Object CimClassName -like Win32* |
  Select-Object CimClassName
```

Y como salida obtenemos lo siguiente:

```
CimClassName
-----
Win32_DeviceChangeEvent
Win32_SystemConfigurationChangeEvent
Win32_VolumeChangeEvent
Win32_SystemTrace
Win32_ProcessTrace
Win32_ProcessStartTrace
Win32_ProcessStopTrace
Win32_ThreadTrace
```

```
Win32_ThreadStartTrace
...
...
```

En los parámetros podemos poner distintos datos. Por ejemplo, para obtener la misma información, pero de un equipo remoto al que tenemos que tener acceso podemos utilizar el parámetro **ComputerName**:

```
Get-CimClass -Namespace root/CIMV2 -ComputerName 192.168.1.75
```

9.4 Obtener información detallada de las clases de WMI

Si ya conocemos el nombre de una clase WMI podemos utilizarlo para obtener información de una forma mucho más detallada.

Por ejemplo, la clase **Win32_OperatingSystem** tiene un montón de propiedades.

Podemos usar **Get-Member** para ver todas las propiedades de una clase.

```
PS > Get-CimInstance -Class Win32_OperatingSystem | Get-Member -
MemberType Property
```

Y la salida sería algo como lo que se muestra a continuación. Donde se ve claramente todas las propiedades y una definición de ellas.

```
TypeName:
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_OperatingSystem
Name                                     MemberType Definition
----
BootDevice                             Property  string BootDevice
{get;}
BuildNumber                             Property  string
BuildNumber {get;}
BuildType                               Property  string BuildType
{get;}
Caption                                 Property  string Caption
{get;}
```


CodeSet	Property	string CodeSet
{get;}		
CountryCode	Property	string
CountryCode {get;}		
CreationClassName	Property	string
CreationClassName {get;}		
CSCreationClassName	Property	string
CSCreationClassName {get;}		
CSDVersion	Property	string CSDVersion
{get;}		
...		

9.5 Visualizar prop. no predeterminadas con *cmdlets* de formato

Existe información en una clase que no se ve de forma predeterminada. Por ejemplo, si queremos ver la información incluida en la clase **Win32_OperatingSystem** que no aparece directamente, podemos mostrarla mediante cmdlets.

```
PS> Get-CimInstance -Class Win32_OperatingSystem |
    Format-Table -Property TotalVirtualMemorySize,
    TotalVisibleMemorySize, FreePhysicalMemory
```

Le estamos indicando las propiedades que queremos que nos muestre.

La salida sería algo como esto:

```
TotalVirtualMemorySize TotalVisibleMemorySize FreePhysicalMemory
-----
33449088          16671872          6451868
```

Podemos formatear la salida también. Por ejemplo, si queremos formatear los datos de memoria como una lista:

```
PS> Get-CimInstance -Class Win32_OperatingSystem | Format-List
Total*Memory*, Free*
```

Y la salida sería la siguiente:

```
TotalVirtualMemorySize : 33449088
```

```
TotalVisibleMemorySize : 16671872
```

FreePhysicalMemory : 6524456

FreeSpaceInPagingFiles : 16285808

FreeVirtualMemory : 18393668

Name : Microsoft Windows 10 Pro | C:\WINDOWS\Device\Harddisk0\Partition2

10 Bibliografía

[1] <https://docs.microsoft.com/es-es/powershell/scripting/overview?view=powershell-7.1>

[2] <https://docs.microsoft.com/es-es/powershell/scripting/windows-powershell/install/windows-powershell-system-requirements?view=powershell-7.1>

[3] PowerShell Core y Windows PowerShell – Los fundamentos del lenguaje [A. Petitjean, R. Lemesle]

[4] <https://blog.netwrix.com/2018/10/04/powershell-variables-and-arrays/>

[5] https://docs.microsoft.com/es-es/powershell/module/microsoft.powershell.core/about/about_variables?view=powershell-7.1#variable-names-that-include-special-characters

[6] <https://www.youtube.com/watch?v=MikI1q2W1BE>

[7] <https://docs.microsoft.com/es-es/powershell/scripting/developer/module/understanding-a-windows-powershell-module?view=powershell-7.1>

[8] <https://docs.microsoft.com/es-es/powershell/module/microsoft.powershell.core/new-modulemanifest?view=powershell-7.1>

[9] <https://docs.microsoft.com/es-es/powershell/scripting/developer/module/how-to-write-a-powershell-script-module?view=powershell-7.1>

[10] <http://freyes.svetlian.com/Modulos.htm>

[11] <https://adamtheautomator.com/powershell-objects/>

[12] <https://www.ediciones-eni.com/open/mediabook.aspx?idR=fe326bf667e4a34b1c241c6310ec963b>

[13] <https://docs.microsoft.com/es-es/powershell/scripting/samples/getting-wmi-objects-get-ciminstance-?view=powershell-7.1>

[14] <https://docs.microsoft.com/es-es/windows/win32/wmisdk/wmi-tasks-for-scripts-and-applications>