

Introducción a Windows PowerShell – Parte I

1. Introducción	3
2. Comenzar a trabajar con PowerShell	3
3. Comandos y cmdlets en PowerShell	4
4. Ayuda en PowerShell	6
5. Redirecciones de salida y tuberías	7
6. Primeros pasos en PowerShell	8
Antes de empezar	8
Escribir un script	9
Ejecutar un script	9
7. Variables en PowerShell	10
Concepto de variable	10
Variables en PowerShell	10
Pedir información al usuario	11
Mostrar información al usuario	11
Tipo de dato de una variable	11
Cómo establecer el tipo de dato de una variable	12
Especificar el tipo de un dato	13
8. Operaciones básicas con variables	15
Operar con variables de tipo texto	15
Operar con variables numéricas	16
Operadores aritméticos especiales	17
Operar con variables lógicas	17
Operadores lógicos	18
Operadores de tipo	19
Variables como objetos	20

9. Variables que guardan múltiples valores	21
Arrays	21
Crear un array	21
Mostrar el contenido de un array	21
Acceder a un elemento individual del array	22
Arrays como objetos	22
Recorrer un array	22
Añadir y quitar valores en un array	24
Tablas hash	26
Mostrar el contenido de una tabla hash	27
Utilizar un elemento individual de la tabla hash	27
Tabla hash como objeto	28
Recorrer una tabla hash	28
Añadir y quitar elementos en una tabla hash	29

1. Introducción

Desde los inicios de MS-DOS, Microsoft ha incluido una **línea de comandos muy básica** que ha ido evolucionando, poco a poco, a lo largo del tiempo, sin perder su compatibilidad con las versiones anteriores. La forma de automatizar acciones en línea de comandos es crear **archivos de procesos por lotes**. Un mecanismo bastante rudimentario y muy limitado que ha ido mostrando sus carencias con el paso del tiempo.

Por ello, en abril de 2006, Microsoft lanzó una **nueva interfaz de consola**, con un gran grupo de órdenes y la capacidad de crear scripts usando una sintaxis diferente, que comparte similitudes con el lenguaje Perl. Esta nueva interfaz comenzó denominándose Monad durante su periodo de desarrollo, pero terminó con el nombre de **PowerShell** en el momento de su lanzamiento.

Siempre ha sido una interfaz gratuita, pero antes de Windows 7 no se incluía con el sistema operativo y necesitaba instalarse a parte. En la actualidad, está incluida en todos los sistemas operativos de Microsoft, aunque requiere la presencia de .NET framework, en el que está basado y del que hereda sus características orientadas a objetos.

Además, en agosto de 2016, Microsoft **publicó el código de PowerShell en GitHub** para que pueda portarse a otros sistemas y actualmente se trabaja en el soporte para GNU/Linux y macOS.

Las ordenes incluidas en PowerShell son muchas, y reciben el nombre de **cmdlets** (de **command-let**). Al principio, PowerShell disponía únicamente de 129 cmdlets básicos. Sin embargo, a lo largo de las versiones, este número se ha ido incrementando. Además, pueden incluirse conjuntos específicos para trabajar con Active Directory, Exchange, u otros roles de servidor.

En la actualidad PowerShell es un entorno enorme, lleno de cmdlets para **administrar casi cualquier característica de Windows y de otras aplicaciones que instalemos sobre él** y, además, **orientado a objetos**.

2. Comenzar a trabajar con PowerShell

Tenemos dos opciones a la hora de ejecutar PowerShell:

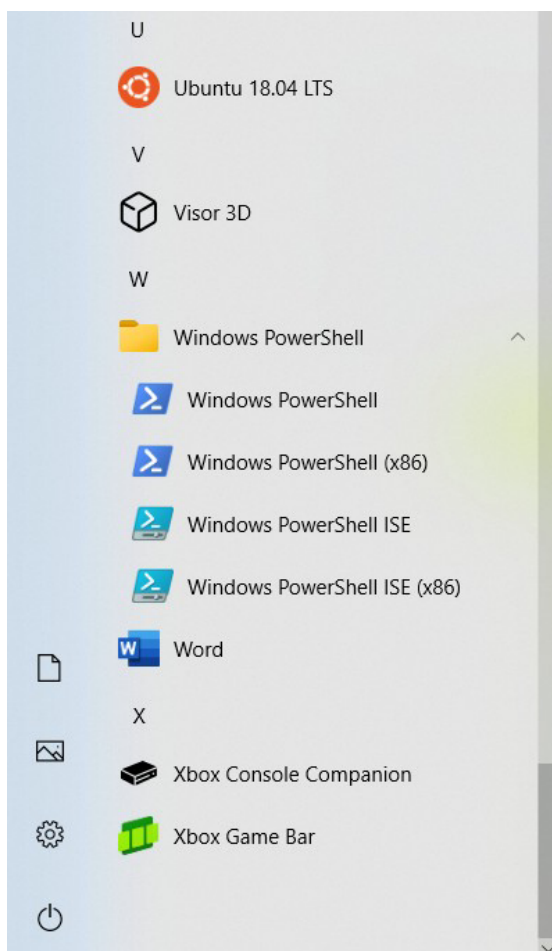
1. Una opción con **entorno gráfico llamada PowerShell ISE** (del inglés, *Integrated Scripting Environment*). Para utilizarla, basta con hacer clic sobre el botón Inicio y desplazarnos por el menú hasta la carpeta **Windows PowerShell**. También tenemos la opción Windows PowerShell ISE (x86). Esta es la versión que elegiremos si nuestro equipo tiene una arquitectura de 32 bits.
2. Una **consola de texto**, similar a la **línea de comandos tradicional**. Para utilizarla comenzaremos por hacer clic sobre el botón Inicio y desplazarnos por el menú hasta la carpeta Windows PowerShell como en el caso anterior. Una vez allí, haremos clic sobre el elemento **Windows PowerShell**. También en este caso disponemos de una opción para equipos con arquitectura de 32 bits.

Para comprobar cómo funciona, podemos aprovechar para conocer el **número exacto de cmdlets** que hay disponibles en nuestro sistema. Para lograrlo, basta con ejecutar la siguiente orden:

```
Get-Command -CommandType cmdlet | Measure-Object

Count      : 548
Average    :
Sum        :
Maximum    :
Minimum    :
Property   :
```

Por último, si necesitamos ejecutar PowerShell con **privilegios administrativos**, basta con hacer clic sobre el elemento Windows PowerShell con el botón derecho del ratón e indicarlo.



3. Comandos y cmdlets en PowerShell

Una de las ventajas inmediatas que encontraremos en PowerShell es que **tendremos a nuestra disposición la mayoría de órdenes que ya conocemos de la línea de comandos tradicional de Windows**. Por ejemplo, PowerShell tiene el **cmdlet Get-ChildItem** para mostrar el **nombre de los archivos y las carpetas que hay en una determinada ruta**. De este modo, escribiendo únicamente el nombre del cmdlet, podré ver los archivos y carpetas del directorio actual.

Sin embargo, seguimos teniendo a nuestra disposición el **comando DIR**, que hacía las mismas funciones en la antigua línea de comandos. Así, escribiendo esa orden, obtendremos un resultado equivalente al anterior. Esto significa que podemos comenzar a utilizar PowerShell con nuestros conocimientos anteriores, en lugar de aprender al principio un número elevado de cmdlets.

Podemos encontrar información sobre los cmdlets más comunes en:

<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/?view=powershell-7.2>

También podemos obtener **información sobre cualquier comando** usando la ayuda del propio PowerShell. Para ellos, sólo tenemos que utilizar el cmdlet **Get-Help**.

Get-Help Get-ChildItem

En realidad, la forma que tiene PowerShell de dar soporte a comandos antiguos es utilizando **alias**.

Podríamos decir que **un alias es un sobrenombre** por el que PowerShell puede conocer a cualquiera de sus cmdlets. Así, en PowerShell **el comando DIR no es más que un alias del cmdlet Get-ChildItem**. De hecho, si nos fijamos en la imagen, la información que ofrece Get-Help sobre Get-ChildItem ya nos informa de que tiene tres alias: **gci, ls, dir**.

The screenshot shows a Windows PowerShell window with the following content:

```

PS C:\Users\34670> Get-ChildItem

Directorio: C:\Users\34670

Mode                LastWriteTime         Length Name
----                -
d-r---         14/10/2020         3:31        3D Objects
d-r---         14/10/2020         3:31        Contacts
d-r---         02/03/2022         1:10        Desktop
d-r---         02/03/2022         1:24        Documents
d-r---         03/03/2022         2:48        Downloads
d-----        26/03/2021        17:31        Ejemplo
d-r---         14/10/2020         3:31        Favorites
d-----        05/10/2020        22:56        knife-workspace
d-r---         14/10/2020         3:31        Links
d-r---         14/10/2020         3:31        Music
dar--l         02/03/2022         0:59        OneDrive
d-r---         27/02/2022         3:15        Pictures
d-r---         14/10/2020         3:31        Saved Games
d-r---         14/10/2020         3:31        Searches
d-----        07/12/2021         3:33        Tracing
d-r---         14/10/2020         3:31        Videos
-a-----        03/06/2021         0:53        42 NULL

PS C:\Users\34670> dir

Directorio: C:\Users\34670

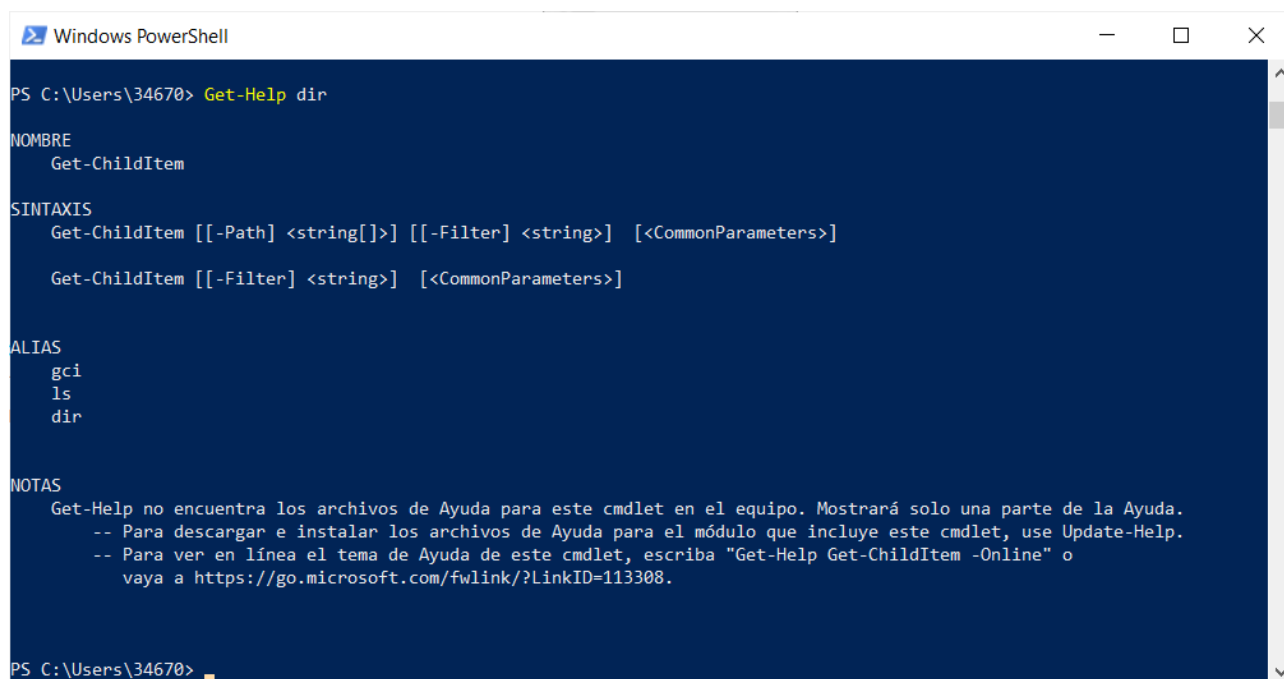
Mode                LastWriteTime         Length Name
----                -
d-r---         14/10/2020         3:31        3D Objects
d-r---         14/10/2020         3:31        Contacts
d-r---         02/03/2022         1:10        Desktop
d-r---         02/03/2022         1:24        Documents
d-r---         03/03/2022         2:48        Downloads
d-----        26/03/2021        17:31        Ejemplo
d-r---         14/10/2020         3:31        Favorites
d-----        05/10/2020        22:56        knife-workspace
d-r---         14/10/2020         3:31        Links
d-r---         14/10/2020         3:31        Music
dar--l         02/03/2022         0:59        OneDrive
d-r---         27/02/2022         3:15        Pictures
d-r---         14/10/2020         3:31        Saved Games
d-r---         14/10/2020         3:31        Searches
d-----        07/12/2021         3:33        Tracing
d-r---         14/10/2020         3:31        Videos
-a-----        03/06/2021         0:53        42 NULL

```

Esta característica podemos utilizarla también cuando conocemos el nombre del alias (el comando antiguo), pero no conocemos el cmdlet correspondiente. En ese caso, utilizaremos Get-Help con el nombre del alias.

Get-Help dir

Los nombres de todos los cmdlets siguen un **patrón común**, formado por un **verbo**, un **guión** y un **nombre en singular**. De este modo será mucho más fácil recordarlos. Además, habitualmente, **se escribe con mayúsculas la primera letra de cada palabra**, pero no se trata más que de una **norma de estilo**, porque PowerShell no distingue entre mayúsculas y minúsculas.



```
Windows PowerShell
PS C:\Users\34670> Get-Help dir

NOMBRE
    Get-ChildItem

SINTAXIS
    Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [<CommonParameters>]

    Get-ChildItem [[-Filter] <string>] [<CommonParameters>]

ALIAS
    gci
    ls
    dir

NOTAS
    Get-Help no encuentra los archivos de Ayuda para este cmdlet en el equipo. Mostrará solo una parte de la Ayuda.
    -- Para descargar e instalar los archivos de Ayuda para el módulo que incluye este cmdlet, use Update-Help.
    -- Para ver en línea el tema de Ayuda de este cmdlet, escriba "Get-Help Get-ChildItem -Online" o
       vaya a https://go.microsoft.com/fwlink/?LinkID=113308.

PS C:\Users\34670>
```

PowerShell dispone también de **autocompletado** de comandos y parámetros (incluidos sus valores). Basta con comenzar a escribir un cmdlet y, cuando pensemos que hemos escrito lo suficiente para que PowerShell lo identifique, pulsamos la tecla **Tab**. En ese momento, PowerShell terminará de escribir el cmdlet por nosotros. A continuación, podremos hacer lo mismo con los parámetros y sus valores.

Si lo que utilizamos es **PowerShell ISE**, al escribir aparecerá una ventana con todos los cmdlets que coincidan con el texto que hemos escrito. Incluso puede aparecer un recuadro con ayuda sobre la sintaxis del cmdlet sugerido en ese momento. Para aceptar la sugerencia y que se complete la escritura, basta con pulsar la tecla Intro.

4. Ayuda en PowerShell

Si nos fijamos en la imagen anterior, **Get-Help** nos avisa de que no está encontrando los archivos de ayuda y sólo se está mostrando una parte. Además, el propio cmdlet nos indica tres formas distintas de obtener la ayuda que nos falta:

1. **Añadir a la sintaxis el argumento -Online**, con lo que Get-Help buscará la información que le falta en la página de Microsoft. Al hacerlo, se abrirá una nueva ventana del navegador web y, en su interior, se mostrará la página con la documentación oficial completa del cmdlet consultado.

```
Get-Help -Online Get-ChildItem
```

2. **Utilizar directamente en el navegador web la URL que nos ofrece Get-Help**. Esto nos llevará exactamente a la misma página web de la imagen anterior.
3. La **instalación de módulos de ayuda de forma local en nuestro equipo**, que debe realizarse desde una ventana de PowerShell con privilegios administrativos, algo que será particularmente útil si no vamos a tener conexión a Internet de manera continua en nuestro equipo.

Para lograrlo, recurriremos al cmdlet **Update-Help**. Sin embargo, si lo utilizamos sin argumentos, como indica la ayuda del propio Get-Help, instalaremos una gran cantidad de información. Por ese motivo es recomendable instalar únicamente los módulos relacionados con PowerShell.

Para lograrlo, sólo tenemos que incluir el **argumento -Module** seguido del nombre de los módulos de ayuda a instalar (en nuestro caso, los relacionados con PowerShell):

```
Update-Help -Module Microsoft.PowerShell*
```

Puede que al final de la instalación, el sistema avise de que algunos de los módulos de ayuda no se ha instalado satisfactoriamente, aunque suele tratarse de módulos poco utilizados.

Si ahora volvemos a utilizar la sintaxis original de Get-Help, obtendremos una salida mucho más detallada que al principio y, como antes, en la parte final de la salida, nos recomiendan otras alternativas:

```
Get-Help Get-ChildItem -Detailed
Get-Help Get-ChildItem -Examples
Get-Help Get-ChildItem -Full
```

5. Redirecciones de salida y tuberías

La mayor parte de las veces, las redirecciones de salida están orientadas a la **obtención de un archivo de texto con la salida que ofrece un cmdlet**. De esta forma, en lugar de consultar el resultado en ese momento, podremos guardarlo para revisarlo en el futuro. Existen dos formas de realizar estas redirecciones:

- **Usando el carácter >:** Crea un nuevo archivo y deposita en él la salida del cmdlet. En caso de que el archivo exista previamente, sustituye su valor anterior por el nuevo.

```
Get-ChildItem > prueba.txt
```

- **Usando el los caracteres >>:** Añade al contenido del archivo la salida del cmdlet. En caso de que el archivo no exista previamente, se crea.

```
Get-ChildItem >> prueba.txt
```

Por otro lado, el objetivo de las **tuberías** consiste en **conectar la salida de un cmdlet con la entrada de otro**, que la tratará como su información de inicio. Este mecanismo no se limita a dos cmdlets, si no que puede haber un tercero, un cuarto, etc. Para establecer una tubería, basta con escribir el primer cmdlet y, a continuación, el carácter |, que también suele llamarse operador de canalización y, por último, escribimos el segundo cmdlet.

En el siguiente ejemplo comenzamos con el cmdlet **Get-Command**, que permite obtener una lista con **todos los comandos instalados en el sistema**, incluidos cmdlets, alias, funciones, flujos de trabajo, filtros, scripts y aplicaciones. Para reducir el abanico, empleamos el **parámetro -CommandType**, donde le indicamos que **sólo estamos interesados en los cmdlets**.

```
Get-Command -CommandType cmdlet | Measure-Object
```

Sin embargo, nuestro objetivo es saber cuántos cmdlets tenemos, no cómo se llaman todos ellos. Por ese motivo, **redireccionamos la salida de Get-Command para que se utilice como entrada en Measure-Object**, cuya función es, básicamente, contar los distintos valores recibidos. Así averiguamos que el número total de cmdlets que teníamos instalados en nuestro sistema era de 548.

A diferencia de otros intérpretes de comandos, PowerShell no comunica un cmdlet con el siguiente enviando la salida en texto plano. En su lugar, **se envían objetos** (en el ejemplo anterior, uno por cada línea de salida con información). De este modo, el cmdlet que recibe la salida puede utilizarla directamente, sin tener que analizar la sintaxis del texto generado por el primer cmdlet. Esto es posible porque **PowerShell se basa en .NET framework, del que hereda sus características orientadas a objetos**.

Incluso **podemos combinar tuberías con redirecciones**, obteniendo el resultado en un archivo. Por ejemplo, para saber, en las carpetas de los usuarios del sistema, qué archivos exceden un tamaño concreto.

Para ello, primero debemos obtener todos los archivos que hay en la carpeta Users con el cmdlet Get-ChildItem indicando la ruta donde buscamos y, en este caso, el parámetro -Recurse (similar al modificador /S en el comando DIR) para que también busque en las subcarpetas:

```
Get-ChildItem C:\Users -Recurse
```

El resultado será una lista de todos los archivos que hay en la ruta indicada. Para **filtrar** los que necesitamos basta con **enviar la salida al cmdlet Where-Object**, y hacerlo en función del tamaño del objeto (en nuestro ejemplo, archivos mayores que 1 MB) y, finalmente, enviaremos la salida a un archivo para poder consultarla más tarde:

```
Get-ChildItem C:\Users -Recurse | Where-Object Length -gt 1mb  
Get-ChildItem C:\Users -Recurse | Where-Object Length -gt 1mb > archivo
```

6. Primeros pasos en PowerShell

Antes de empezar

Un aspecto importante para PowerShell es la **seguridad**. Su objetivo es evitar que se ejecuten, sin la autorización del usuario, **scripts que puedan dañar al equipo**. Y para alcanzarlo, dispone de varios **modos de restricción en la ejecución** de scripts.

Lo primero es que, **de forma predeterminada, el sistema no permite la ejecución de ningún tipo de scripts**. De ese modo, un usuario sin los conocimientos adecuados, no tendrá que preocuparse de nada. Simplemente, su sistema no ejecuta scripts. No obstante, un usuario que sí pretenda usar scripts en su actividad diaria, deberá **ajustar esta política predeterminada del sistema**. Pero antes, necesitamos saber que existen cuatro niveles de permisividad disponibles:

- **Restricted:** En este nivel no se permite la ejecución de scripts. Es decir, PowerShell sólo puede utilizarse en modo interactivo. Esta es la **opción predeterminada**.
- **AllSigned:** Cuando esta sea la opción elegida, deberán estar autenticados todos los scripts, antes de poder ejecutarlos. Es la opción más **restrictiva**.
- **RemoteSigned:** En este caso, sólo deberán estar autenticados los scripts que procedan de una ubicación remota. Por ejemplo, los que hayan sido **descargados**.
- **Unrestricted:** Si elegimos esta opción, se ejecutará cualquier script sin importar su origen. Se trata de la **opción menos recomendada**.

Para conocer la configuración actual de la política de ejecución de scripts, ejecutaremos el siguiente cmdlet:

```
Get-ExecutionPolicy
```

Para establecer la política de ejecución de scripts, necesitaremos ejecutar el **cmdlet Set-ExecutionPolicy** seguido del **nombre de política elegido**. Para ejecutar este cmdlet necesitaremos privilegios administrativos.

```
Set-ExecutionPolicy RemoteSigned
```

Al ejecutarlo, nos aparece un mensaje avisándonos del peligro que supone el cambio en la política de ejecución de scripts. A continuación, deberemos indicar si estamos de acuerdo con el cambio. Incluso podemos pulsar el **carácter ?** para obtener ayuda.

Escribir un script

Un **script de PowerShell** sólo es un **archivo de texto plano** que contiene la **secuencia de órdenes necesarias para automatizar la tarea** que queramos. Normalmente, en cada línea del archivo aparecerá un cmdlet diferente y éstos se ejecutarán por orden de aparición. Es decir: se ejecutará el cmdlet de la primera línea, después el de la segunda y así sucesivamente.

Para que el archivo de texto sea tratado como un script de PowerShell, sólo es necesario que tenga la **extensión .ps1**.

Para este primer script realizaremos algo sencillo: escribir dos líneas de texto en la terminal. En la primera línea aparecerá el texto *Hola Mundo* y en la segunda *Esto es PowerShell*.

Para conseguirlo, sólo tenemos que escribir lo siguiente en el editor:

```
Write-Host "Hola Mundo"  
Write-Host "Esto es PowerShell"
```

Cuando hayamos terminado, simplemente cerramos la ventana.

Ejecutar un script

Para ejecutar un script, sólo hay que escribir su nombre en el prompt de PowerShell. En ese momento, el sistema buscará, en su ruta predeterminada, un script con ese nombre. Y, si lo encuentra, lo ejecutará. Sin embargo, es posible que recibamos un error porque el directorio donde se encuentra el script no está entre los directorios predeterminados donde busca el sistema. Por ese motivo, al nombre del script debemos anteponerle la **ruta donde encontrarlo**.

```
C:\Users\usuario\ejemplo01.ps1
```

Si la ruta en cuestión coincide con la ruta actual en el momento de ejecutarlo, podemos simplificarlo:

```
.\ejemplo01.ps1
```

Para saber cuál es la ruta predeterminada del sistema, basta con escribir la siguiente orden:

```
$env:Path
```

```
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Users\usuario\AppData\Local\Microsoft\WindowsApps;
```

Observando detenidamente vemos que, en realidad, se trata de varias rutas separadas por el carácter de punto y coma.

```
C:\Windows\system32;  
C:\Windows;  
C:\Windows\System32\Wbem;  
C:\Windows\System32\WindowsPowerShell\v1.0\;  
C:\Users\usuario\AppData\Local\Microsoft\WindowsApps;
```

Bastaría con guardar el script en cualquiera de esos directorios para que pudiésemos eliminar la ruta del nombre del archivo. O bien, agregar una ruta propia modificando el contenido de la variable.

```
Set-Item -Path Env:Path -Value ($Env:Path + ";C:\Temp")
```

https://docs.microsoft.com/es-es/powershell/module/microsoft.powershell.core/about/about_environment_variables?view=powershell-7.2

7. Variables en PowerShell

Concepto de variable

Podemos decir que una variable es una porción de memoria principal a la que ponemos un nombre que facilite su identificación y manejo. Su objetivo consiste en **permitir el almacenamiento de un valor** en particular para su **uso posterior a lo largo del script**.

El nombre de una variable permanecerá inalterable a lo largo del script, y se recomienda que describa el tipo de información que contiene. Por su parte, el valor que almacenemos dentro de la variable podrá cambiar a lo largo del script, según las necesidades de la tarea que estemos resolviendo.

Cuando hablamos del espacio que ocupan las variables en memoria, tenemos dos opciones:

- **Longitud fija:** El tamaño de la variable en la memoria no cambiará aunque cambie el dato que almacena. En este grupo tenemos la mayoría de los tipos de **datos básicos**: *char*, *byte*, *int*, *long*, *single*, *double*, *decimal*, *datetime* y *bool*.
- **Longitud variable:** En este caso, el tamaño de **la variable se ajusta a dato** que le estemos asignando. En este grupo se encuentran: *string*, *array*, *hashtable* y *xml*.

Variables en PowerShell

Para definir una variable en PowerShell sólo tenemos que nombrarla siguiendo una serie de restricciones:

- El primer carácter debe ser siempre un símbolo de dólar (\$).
- Después, podemos utilizar cualquier combinación de letras, números o símbolos.
- También pueden utilizarse espacios en blanco pero, en este caso, el nombre debe ir rodeado por símbolos de llaves ({}).

Habitualmente, la primera vez que se utiliza una variable es para asignarle un valor inicial. Por ejemplo:

```
$hola = ";Hola Mundo!"
```

El símbolo de igual (=) permite asignar valores en PowerShell, y debemos leerlo de derecha a izquierda. Es decir, el valor representado a la derecha del símbolo es asignado a la variable que aparezca a la izquierda.

En PowerShell, los valores textuales van rodeados por el símbolo de comillas (""). El primero indica dónde comienza el texto y el segundo dónde acaba. También pueden utilizarse símbolos de apóstrofe ('), pero no pueden mezclarse, es decir, debemos acabar con el mismo símbolo con el que comencemos.

Algunos ejemplos de nombre válidos para variables podrían ser estos:

```
$Precio  
$Resultado_1  
${Resultado 1}
```

Esta forma de definir variables es **implícita**, porque **la variable se define automáticamente al utilizarla**. Sin embargo, existe también un **modo explícito usando el cmdlet New-Variable**. Por ejemplo, podemos definir una variable usando la siguiente sintaxis:

```
New-Variable Resultado_1
```

En este caso, no hemos utilizado el símbolo de dólar (\$) al principio, pero sí lo incluiremos al utilizarla. Si queremos una sintaxis más descriptiva, podemos incluir argumentos e incluso podríamos asignar un valor inicial:

```
New-Variable -Name Resultado_1  
New-Variable -Name Resultado_1 -Value 200
```

Si una variable va a contener un valor que no debe cambiar a lo largo del script, podemos indicar que sea de sólo lectura, es decir, una **constante**.

```
New-Variable -Name Resultado_1 -Value 200 ReadOnly
```

En cualquier momento podemos obtener una lista completa de las variables que se han definido hasta el momento utilizando el cmdlet **Get-Variable**.

<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/new-variable?view=powershell-7.2>

Pedir información al usuario

Uno de los primeros usos que daremos a las variables será el de recibir y almacenar algún valor que se solicite al usuario y el cmdlet que se encarga de **pedir información al usuario es Read-Host**. Su funcionamiento consiste en **mostrar en pantalla un texto** (de forma **opcional**) y **guardar en una variable**, también como texto, lo **que escriba el usuario**.

```
$Nombre = Read-Host "¿Cómo te llamas?"
```

Al ejecutar la línea anterior, el usuario verá en pantalla el texto *¿Cómo te llamas?* En respuesta, escribirá su nombre que será guardado en la **variable \$Nombre**.

Mostrar información al usuario

Una vez que hemos visto cómo pedir **información** al usuario, el siguiente paso es **ofrecerla**. Esto lo conseguimos con el **cmdlet Write-Host**. De este modo, para saludar al usuario que ha escrito su nombre en el apartado anterior, podríamos escribir:

```
Write-Host "Bienvenido"  
Write-Host $Nombre
```

En PowerShell, **para mostrar información, podemos omitir el nombre del cmdlet** de modo que, si en una línea del script ponemos únicamente el nombre de una variable, se mostrará su valor igual que si hubiésemos escrito delante Write-Host.

Tipo de dato de una variable

En los ejemplos anteriores hemos visto únicamente **cómo podemos guardar texto** en una variable. Sin embargo **son muchos los tipos de datos que pueden contener**. En la tabla siguiente vemos los más habituales:

Tipos de datos		
Tipo de dato	Descripción	Rango de valores
[string]	Cadena de caracteres Unicode	Depende del contenido
[char]	Un sólo carácter Unicode de 16 bits	-32768 a 32767

[byte]	Un sólo carácter Unicode de 8 bits	0 a 255
[int]	Entero con signo de 32 bits	-2147483648 a 2147483647
[long]	Entero con signo de 64 bits	-9223372036854775808 a 9223372036854775807
[single]	Número en coma flotante de 32 bits (precisión simple)	Rango negativo -3.402823E38 a -1.401298E-45 Rango positivo 1.401298E-45 a 3.402823E38
[double]	Número en coma flotante de 64 bits (precisión doble)	Rango negativo -1.79769313486232E308 a 4.94065645841247E-324 Rango positivo 4.94065645841247E-324 a 1.79769313486232E308
[decimal]	Valores enteros de 128 bits donde una parte del número es decimal	-79228162514264337593543950335 a 79228162514264337593543950335
[datetime]	Fecha y hora	1 enero 100 hasta 31 diciembre 9999
[bool]	Valor lógico	True o False
[array]	Objeto que representa una matriz de valores	Depende del contenido
[hashtable]	Objeto que representa una tabla hash	Depende del contenido
[xml]	Objeto con contenido XML	Depende del contenido

Si necesitamos saber el tipo de dato al que pertenece una determinada variable, puedes utilizar una expresión como esta:

```
Write-Host $Nombre.GetType().Name
```

Si lo aplicamos al ejemplo anterior, obtendríamos la salida: **String**.

Si la variable es **numérica**, también podríamos obtener su **rango de valores** usando expresiones como:

Para su valor mínimo: `[decimal]::MinValue`

Para su valor máximo: `[decimal]::MaxValue`

Para otros tipos de datos numéricos, sólo hay que sustituir **[decimal]** por el nombre del tipo adecuado.

Cómo establecer el tipo de dato de una variable

En PowerShell existen **dos formas de establecer la naturaleza del dato** que va a almacenar una variable:

- **Implícita:** El tipo de dato que puede almacenar la variable se establece en función del valor asignado.
- **Explícita:** El programador establece el tipo de dato que podrá contener la variable. De este modo, cuando asignemos un valor que no coincide con el tipo especificado, PowerShell tratará de amoldarlo a la variable.

En la **definición implícita**, el tipo de dato de estas variable **se establece en el momento de asignarle valor**. En este caso, la **variable \$hola** se crea en el momento de usarla y se le asigna un valor de tipo texto (al que llamaremos **string** o cadena de caracteres).

```
$hola = ";Hola Mundo!"
```

Este método también se utiliza para crear datos de otros tipos. Por ejemplo, en el siguiente ejemplo creamos una variable numérica (en particular, de tipo **double**):

```
$precio = 4.99
```

Una variable, cuya definición de tipo se ha realizado de forma implícita, podrá ir cambiando el tipo de dato almacenado según nuestras necesidades. Por ejemplo, después de la línea anterior podríamos escribir lo siguiente **sin experimentar ningún tipo de error**:

```
$precio = "Justo"
```

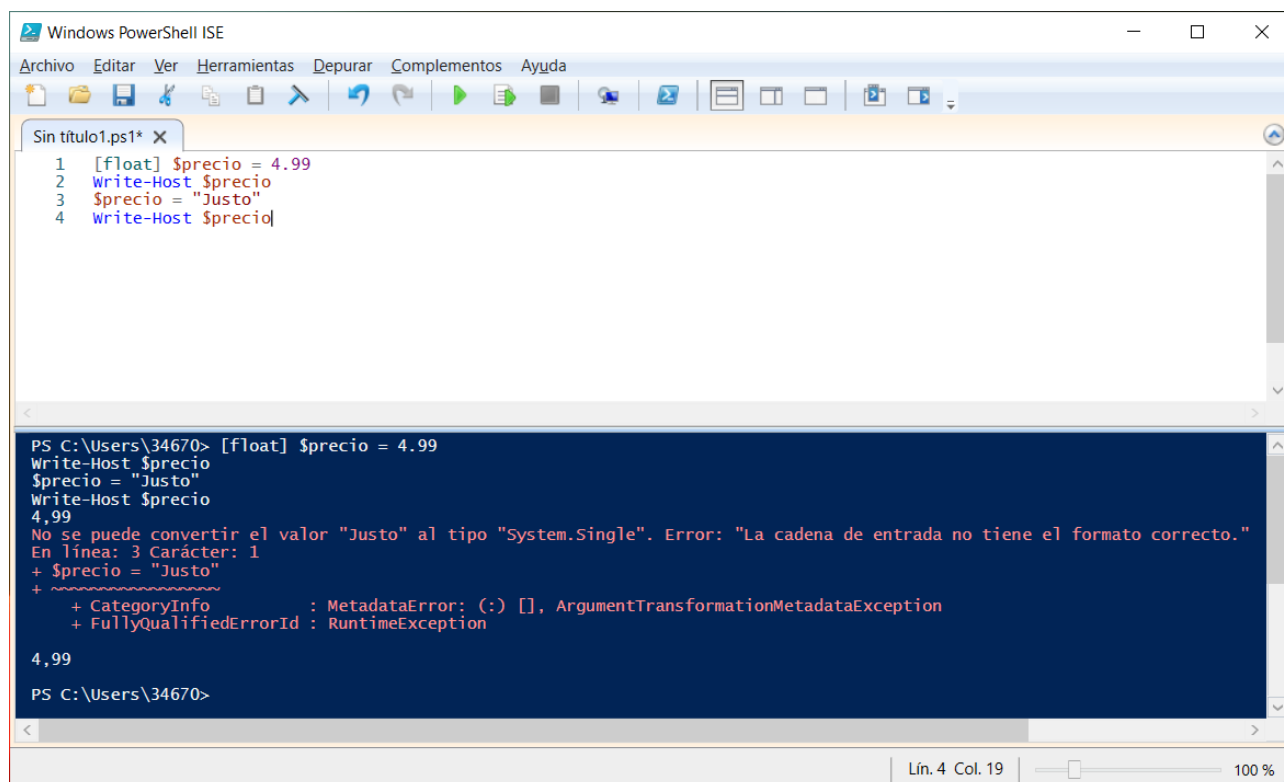
En la **definición explícita** informamos a PowerShell del tipo de dato que puede almacenar una variable particular. Al hacerlo, estamos **estableciendo un límite en cuanto a la naturaleza de los datos** que pueden almacenarse dentro de la variable.

Por ejemplo, como hemos dicho antes, la variable \$precio del ejemplo anterior será de **tipo double, por lo que ocupará en memoria 64 bits**. Si nos parece excesivo este gasto de memoria por la naturaleza de los datos que estamos almacenando, al definir la variable, podemos especificar el tipo de dato al que pertenece.

```
[float] $precio = 4.99
```

Cuando el dato asignado **no coincida con el tipo esperado**, pueden ocurrir dos cosas:

- Que las **características del dato se modifiquen** para amoldarse a la variable.
- Cuando lo anterior no es posible, se producirá un **error**.



The screenshot shows the Windows PowerShell ISE interface. The script editor contains the following code:

```
1 [float] $precio = 4.99
2 Write-Host $precio
3 $precio = "Justo"
4 Write-Host $precio
```

The console output shows the execution of the script:

```
PS C:\Users\34670> [float] $precio = 4.99
Write-Host $precio
4,99
$precio = "Justo"
Write-Host $precio
No se puede convertir el valor "Justo" al tipo "System.Single". Error: "La cadena de entrada no tiene el formato correcto."
En línea: 3 Carácter: 1
+ $precio = "Justo"
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException

4,99
PS C:\Users\34670>
```

Especificar el tipo de un dato

Otra alternativa que tenemos a nuestra disposición consiste en indicar **el tipo al que pertenece el propio dato**. De este modo, podríamos lograr las ventajas de las dos opciones anteriores: tener una variable que

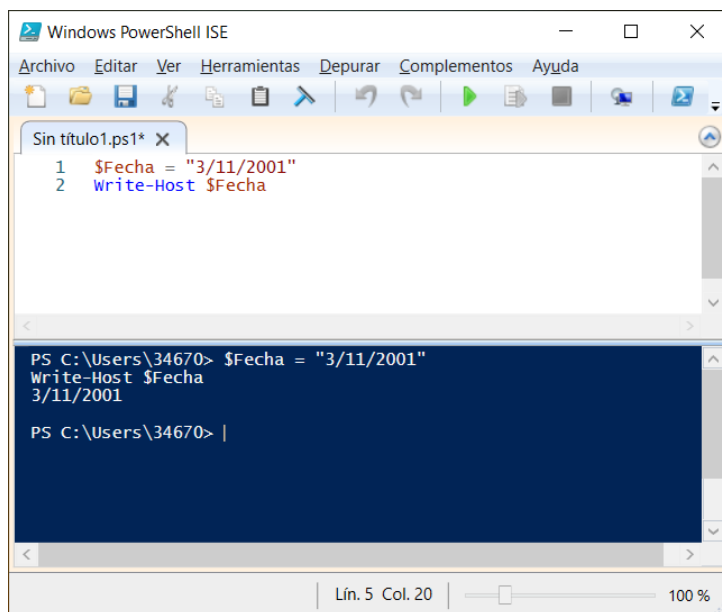
pueda cambiar su tipo y **usar el espacio de memoria adecuado para el dato que estamos guardando**. Para hacerlo, bastaría con escribir lo siguiente:

```
$precio = [float] 4.99
```

Una ventaja añadida es que podemos controlar, de forma estricta, el tipo de valor almacenado.

```
$Fecha = "3/11/2001"
```

Aunque para nosotros el dato anterior es claramente una fecha, para PowerShell no es más que un texto. De hecho, si mostramos el contenido de variable, obtenemos exactamente lo que asignamos:



```
Windows PowerShell ISE
Sin título1.ps1* X
1 $Fecha = "3/11/2001"
2 Write-Host $Fecha

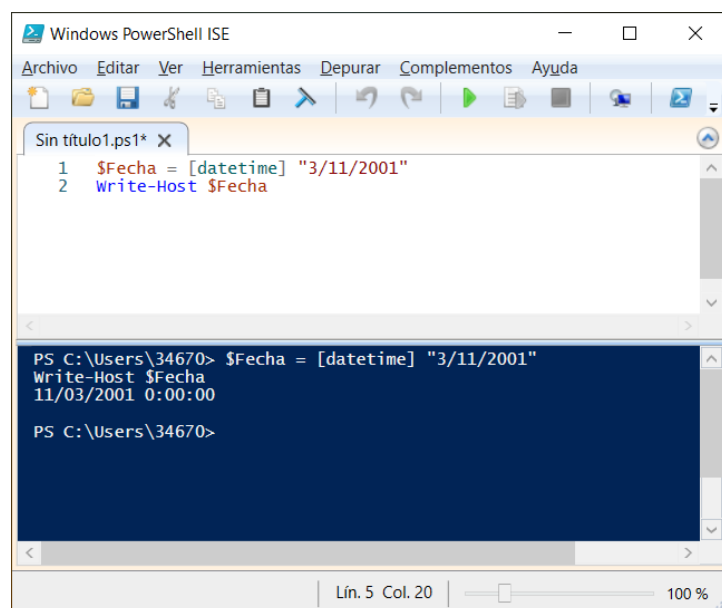
PS C:\Users\34670> $Fecha = "3/11/2001"
Write-Host $Fecha
3/11/2001

PS C:\Users\34670> |
```

Sin embargo, cuando especificamos el tipo de dato, éste será asumido por la variable y PowerShell entenderá que se trata de una fecha:

```
$Fecha = [datetime] "3/11/2001"
```

La diferencia se aprecia claramente cuando consultamos el contenido de la variable:



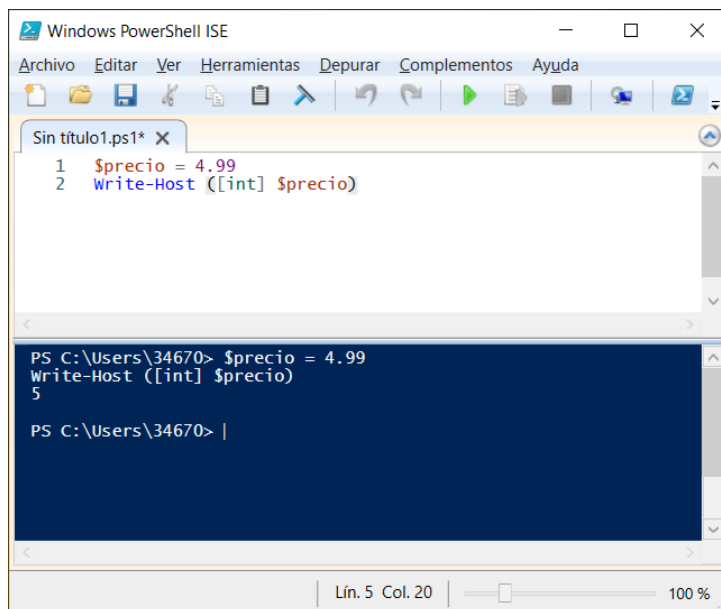
```
Windows PowerShell ISE
Sin título1.ps1* X
1 $Fecha = [datetime] "3/11/2001"
2 Write-Host $Fecha

PS C:\Users\34670> $Fecha = [datetime] "3/11/2001"
Write-Host $Fecha
11/03/2001 0:00:00

PS C:\Users\34670>
```

Cuando amoldamos un dato a un tipo diferente puede ocurrir que **la coincidencia no sea exacta**. En este caso, PowerShell tratará de buscar la solución más adecuada. Por ejemplo, aquí nos ofrece el entero más próximo al valor dado.

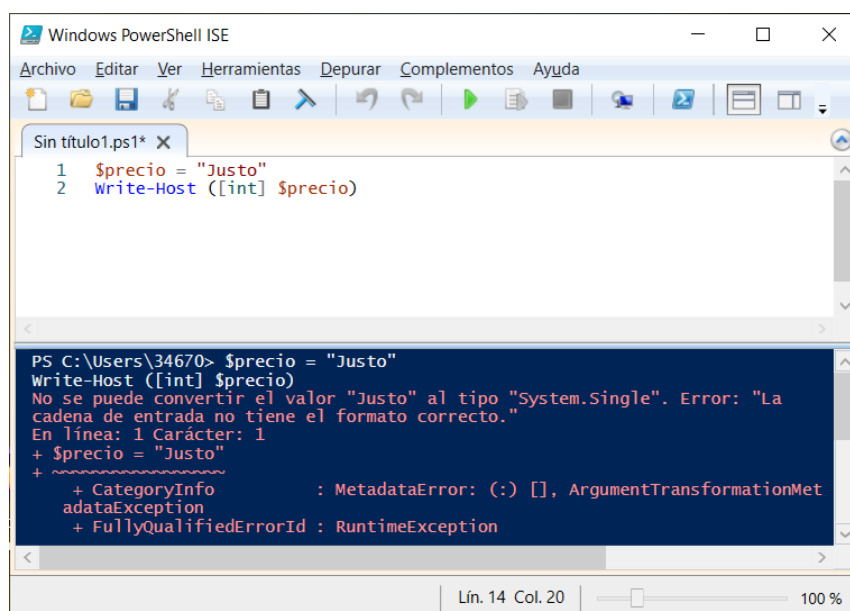
```
$precio = 4.99  
Write-Host ([int] $precio)
```



```
Sin título1.ps1* X  
1 $precio = 4.99  
2 Write-Host ([int] $precio)  
  
PS C:\Users\34670> $precio = 4.99  
Write-Host ([int] $precio)  
5  
  
PS C:\Users\34670> |
```

Cuando esto no sea posible, sencillamente se producirá un error de ejecución.

```
$precio = "Justo"  
Write-Host ([int] $precio)
```



```
Sin título1.ps1* X  
1 $precio = "Justo"  
2 Write-Host ([int] $precio)  
  
PS C:\Users\34670> $precio = "Justo"  
Write-Host ([int] $precio)  
No se puede convertir el valor "Justo" al tipo "System.Single". Error: "La  
cadena de entrada no tiene el formato correcto."  
En línea: 1 Carácter: 1  
+ $precio = "Justo"  
+ ~~~~~  
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMet  
adataException  
+ FullyQualifiedErrorId : RuntimeException
```

8. Operaciones básicas con variables

Operar con variables de tipo texto

El operador más sencillo que podemos usar con variables de texto es el **operador de asignación**.

Con él, **asignamos el contenido de la expresión que haya a su derecha a la variable indicada a la izquierda**. Además, podemos utilizarlo con variables de cualquier tipo.

Otra de las operaciones básicas con variables de tipo texto es unir o **concatenar** el contenido de diferentes variables en una sola. Para lograrlo, podemos utilizar **el operador de suma (+)**.

```
$nombre = "Camilo"
$saludo = "Hola"
$mensaje = $saludo + $nombre
Write-Host $mensaje
```

En este caso, el mensaje aparece **junto**, sin ningún tipo de separación entre el contenido de la variable **\$nombre** y el de **\$saludo**. Esto es porque **PowerShell no interpreta el sentido de lo que estamos haciendo**. Únicamente se limita a pegar el contenido de ambas variables en una sola. Para solucionarlo, podríamos crear una expresión como esta:

```
$mensaje = $saludo + " " + $nombre
```

Sin embargo, PowerShell también incorpora la **capacidad de sustituir el contenido de una variable cuando ésta forma parte de un texto**. Es decir, en el ejemplo anterior podríamos asignar a **\$mensaje** un texto que contenga las dos variables que necesitamos y el resultado obtenido sería equivalente.

```
$mensaje = "$saludo $nombre"
```

Por último, indicar que no siempre es necesario asignar el resultado de una expresión a una variable. Por ejemplo, en los ejemplos anteriores podríamos haber escrito directamente:

```
Write-Host "$saludo $nombre"
Write-Host ($saludo + " " + $nombre)
```

Operar con variables numéricas

Los operadores que pueden utilizarse con variables numéricas reciben el nombre de **operadores aritméticos** y tenemos los siguientes:

Operadores aritméticos		
Operador	Función	Ejemplo
+	Suma	\$resultado = \$x + 3
-	Resta	\$resultado = \$x - 3
*	Multipliación	\$resultado = \$x * 3
/	División	\$resultado = \$x / 3
%	Resto	\$resultado = \$x % 3

De ellos, probablemente el único más complejo sea el último. Se trata de un operador que **calcula la división entera** (sin obtener decimales) entre un dividendo (en este caso el contenido de la variable \$x) y un divisor (en este caso, 3) y, en lugar de devolver el cociente de la división, **devuelve el resto**.

```
$x = 17
$resultado = $x % 3
Write-Host $resultado
```

Y cuando lo ejecutemos, **obtendremos**, por tanto, **el valor 2**. Ya que el cociente de la división entera sería 5 y quedaría un resto de 2.

Operadores aritméticos especiales

Cuando la **variable que participa en la operación y la que recibe el resultado son la misma**, PowerShell dispone de una serie de operadores que nos permiten **resumir las expresiones**. Por ejemplo, si queremos incrementar en tres unidades el valor contenido en la **variable \$x**, podríamos escribir las expresiones:

```
$x = $x + 3
$x += 3
```

Y esto podríamos aplicarlo a todos los operadores aritméticos.

Operadores aritméticos especiales			
Operador	Significado	Ejemplo	Equivalencia
+=	Incrementa el valor de la variable	<code>\$x += 3</code>	<code>\$x = \$x + 3</code>
-=	Reduce el valor de la variable	<code>\$x -= 3</code>	<code>\$x = \$x - 3</code>
*=	Multiplica el valor de la variable	<code>\$x *= 3</code>	<code>\$x = \$x * 3</code>
/=	Divide el valor de la variable	<code>\$x /= 3</code>	<code>\$x = \$x / 3</code>

Todavía existe un **caso especial** que puede escribirse de **forma aún más reducida**. Es el caso en que debamos **incrementar o decrementar el contenido de la variable en uno**, que podemos escribir así:

```
$x = $x + 1
$x += 1
$x++
```

Y lo mismo podemos hacer con la resta:

Operadores aritméticos especiales			
Operador	Significado	Ejemplo	Equivalencia
++	Incrementa en uno el valor de la variable	<code>\$x++</code>	<code>\$x = \$x + 1</code>
--	Reduce en uno el valor de la variable	<code>\$x--</code>	<code>\$x = \$x - 1</code>

Operar con variables lógicas

Las variables de **tipo lógico [bool]** únicamente pueden recibir dos tipos de valores: **verdadero (\$true)** o **falso (\$false)**. Este valor podemos **asignarlo de forma directa** o como **resultado de una expresión**. Para asignar el valor de forma directa, basta con escribir algo como:

```
$salir = $false
```

Sin embargo, **para crear una expresión lógica**, necesitaremos una serie de operadores: los **operadores lógicos**. Con ellos podemos averiguar, por ejemplo, si dos variables tienen el mismo valor:

```
$iguales = $a -eq $b
```

En este caso, la variable **\$iguales** recibirá el valor **\$true** si el contenido de la **variable \$a** es igual que el **contenido de la variable \$b**. En caso contrario, recibirá el valor **\$false**.

Operadores de comparación		
Operador	Significado	Ejemplo (\$true)
-eq	Igual	<code>1 -eq 1</code>

-ieq	Igual Pero con valores de tipo texto no tendrá en cuenta la diferencia entre mayúsculas y minúsculas	"hola" -ieq "HOLA"
-ceq	Igual Con valores de tipo texto, tendrá en cuenta la diferencia entre mayúsculas y minúsculas	"HOLA" -ceq "HOLA"
-ne	Diferente	3 -ne 5
-lt	Menor que	3 -lt 5
-le	Menor o igual	5 -le 5 3 -le 5
-gt	Mayor que	5 -gt 3
-ge	Mayor o igual	5 -ge 5 5 -ge 3

Además de los operadores básicos de la tabla anterior, disponemos de algunas **opciones más avanzadas para trabajar con texto y colecciones de valores**. Por ejemplo, podemos utilizar **caracteres comodín** para saber si un determinado texto se encuentra dentro de otro:

```
$encontrado = $nombre -like Fer
```

La variable **\$encontrado** recibe el **valor \$true** sólo si el contenido de la **variable nombre comienza por las letras Fer**.

Los operadores de este tipo son:

Operadores de comparación		
Operador	Significado	Ejemplo (\$true)
-like	Es como	"Fermín" -like "Fer*"
-notlike	No es como	"Fermín" -notlike "Fern*"
-contains	Contiene	9,5,2 -contains 5
-notcontains	No contiene	9,5,2 -notcontains 1

Operadores lógicos

Los operadores lógicos nos permiten **unir varias expresiones condicionales en una sola**. Por ejemplo, si disponemos de **tres variables numéricas y necesitamos saber si una de ellas contiene el valor mayor**, podríamos escribir algo similar a lo que aparece a continuación.

Aquí, la **variable \$mayor** recibirá el valor **\$true** sólo cuando el valor de **\$a sea superior al de \$b y \$c**.

```
$mayor = ($a -gt $b) -and ($a -gt $c)
```

En PowerShell tenemos los siguientes operadores lógicos:

Operadores lógicos		
Operador	Descripción	Ejemplo (\$true)
-and	Devuelve \$true cuando las dos expresiones que intervienen devuelven el valor \$true	(5 -ge 3) -and (5 -le 5)

-or	Devuelve \$true cuando alguna de las dos expresiones que intervienen devuelve el valor \$true	(5 -gt 3) -or (5 -lt 5)
-xor	Devuelve \$true cuando sólo una de las expresiones que intervienen devuelve el valor \$true	(5 -gt 3) -xor (8 -le 5)
-not !	Devuelve \$true cuando la expresión sobre la que actúa devuelve el valor \$false	-not (5 -lt 3) !(5 -lt 5)

Los primeros tres operadores son binarios. Es decir, obtienen su resultado a partir de dos expresiones de comparación distintos. Por el contrario, **el último operador es unario**, porque **actúa sobre una sola expresión** de comparación, en concreto para **invertir su resultado**. Además, es el único que puede escribirse de dos formas diferentes: siguiendo la misma estructura que el resto de operadores (**-not**) o sustituyendo el operador por un signo de admiración (**!**). Esta última sintaxis tiene su origen en el lenguaje C y actualmente se utiliza en muchos lenguajes de programación.

Vamos a comprobar los resultados de los ejemplos anteriores:

```

1 Write-Host ((5 -ge 3) -and (5 -le 5))
2 Write-Host ((5 -gt 3) -or (5 -lt 5))
3 Write-Host ((5 -gt 3) -xor (8 -le 5))
4 Write-Host (-not (5 -lt 3))
5 Write-Host (!(5 -lt 5))

PS C:\Users\34670> Write-Host ((5 -ge 3) -and (5 -le 5))
Write-Host ((5 -gt 3) -or (5 -lt 5))
Write-Host ((5 -gt 3) -xor (8 -le 5))
Write-Host (-not (5 -lt 3))
Write-Host (!(5 -lt 5))
True
True
True
True
True

PS C:\Users\34670> |

```

Operadores de tipo

Este tipo de operadores permite comprobar **si una variable (o cualquier otro valor) se corresponde con un tipo de dato en particular**. En el ejemplo que vemos a continuación, la **variable \$esFecha** recibe el valor **\$false**, porque el dato, aunque parece una fecha, en realidad es un texto (su tipo de dato es **string**).

```

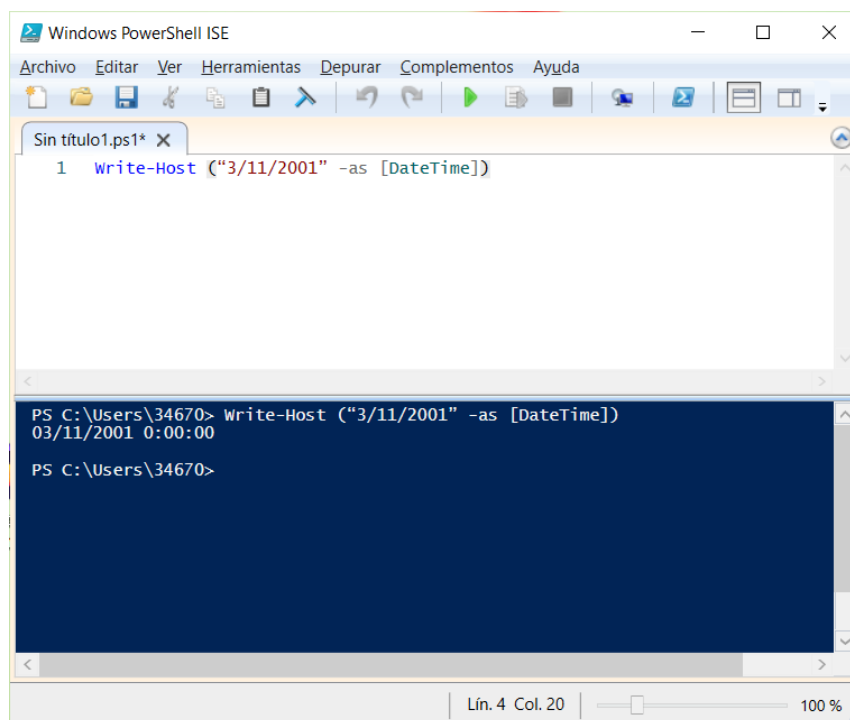
$fecha = "3/11/2001"
$esFecha = $fecha -is [DateTime]

```

Operadores lógicos		
Operador	Descripción	Ejemplo (\$true)
-is	Devuelve \$true cuando el dato se corresponde con el valor indicado	"3/11/2001" -is [string]
-isnot	Devuelve \$true cuando el dato no se corresponde con el valor indicado	"3/11/2001" -isnot [DateTime]

En PowerShell también disponemos del **operador -as**, que **trata de convertir el dato que se encuentre a su izquierda, al tipo que indiquemos a la derecha**. Cuando la conversión no sea posible, se producirá un error de ejecución. Por ejemplo:

```
Write-Host ("3/11/2001" -as [DateTime])
```



Igual que nos ocurría al especificar el tipo de un dato, puede ocurrirnos que **la conversión no sea exacta**. Como entonces, **PowerShell tratará de buscar la solución más adecuada**. Sin embargo, cuando esto no sea posible, **sencillamente no hará nada**.

Variables como objetos

PowerShell está basado en .NET Framework, lo que significa que en la práctica, **nuestros scripts tendrán acceso a todo el modelo de objetos subyacentes en .NET Framework**. De hecho, cada variable que definimos en PowerShell es en realidad un **objeto de .NET Framework**. Y, como tal, **tendrá asociados una serie de métodos que podremos utilizar para realizar ciertas acciones sin tener que escribir ningún código complementario**.

Podríamos decir que un método es un **fragmento de código, asociado a un determinado objeto** (en nuestro caso una variable o un dato), que tiene una **función concreta**.

Un ejemplo podría ser averiguar la posición en la que aparece la primera letra “o” en una palabra contenida en una variable.

```
$hola = ";Hola Mundo!"
```

La solución más sencilla es recurrir al método `IndexOf`, que nos permite buscar un determinado texto dentro del objeto en el que lo utilizemos:

```
Write-Host $hola.IndexOf("o")
```

El resultado es 2 y no 3. El motivo es que **las posiciones de los caracteres se cuentan a partir de cero**.

Otro ejemplo podría ser obtener el contenido de la variable anterior, pero convertido en minúsculas. En realidad, **cada tipo de datos tiene multitud de métodos a los que podemos recurrir**.

```
Write-Host $hola.ToLower()
```

Los métodos para, por ejemplo, los tipos **string** o **int** serían los siguiente:

<https://docs.microsoft.com/en-us/dotnet/api/system.string?redirectedfrom=MSDN&view=net-6.0#Methods>
<https://docs.microsoft.com/en-us/dotnet/api/system.int32?redirectedfrom=MSDN&view=net-6.0#Methods>

9. Variables que guardan múltiples valores

Arrays

Básicamente, un **array es una lista de valores de cualquier tipo**, que podemos **identificar de forma individual por la posición que ocupan** dentro de la lista. Las posiciones de los diferentes elementos se numeran a partir de cero.

Crear un array

A diferencia de lo que se hace en otros lenguajes de programación, en PowerShell no hay que definir una variable de forma diferente para que actúe como un array. Es suficiente con **asignarle un grupo de valores**, siguiendo este formato:

```
$variable = @(valor1, valor2, valor3)  
$a = @(1, 2, 3, 4)
```

Este es el formato explícito. Sin embargo, también sería correcto usar expresiones como estas:

```
$a = 1, 2, 3, 4  
$a = 1..4
```

Además, **cada elemento de un array puede ser de un tipo diferente** aunque, si necesitamos restringir el tipo de dato que podemos almacenar en ella, basta con indicarlo delante del nombre de la variable. En ese caso, seguiremos el siguiente formato:

```
[int[]] $a = @(1, 2, 3, 4)
```

Aunque se ha utilizado el formato estándar en el ejemplo, la definición explícita de tipo es válida también con la sintaxis reducida anterior:

```
[int[]] $a = 1, 2, 3, 4  
[int[]] $a = 1..4
```

Incluso podemos crear un array vacío al que ir añadiendo elementos más tarde:

```
$a = @()
```

Mostrar el contenido de un array

Si queremos ver en pantalla el contenido de todo el array en un momento dado, podemos utilizar la misma técnica que con una variable simple:

```
$a = @(1, 2, 3, 4)  
Write-Host $a
```

Acceder a un elemento individual del array

Para acceder a un **elemento concreto del array**, basta con **escribir su nombre seguido de la posición** que necesitamos encerrada entre corchetes. Por ejemplo, para mostrar el primer elemento del array anterior, escribiríamos esto:

```
$a = @(1, 2, 3, 4)
Write-Host $a[0]
```

Aunque, si lo que queremos es utilizar el elemento dentro de una expresión, nos limitamos a utilizar la notación anterior como si se tratara de una variable simple. Por ejemplo, para incrementar en uno el valor del elemento.

```
$a[0] = $a[0] + 1
$a[0]++
```

Arrays como objetos

Como ya sabemos, internamente, PowerShell trata a todas las variables como objetos y, gracias a esta característica, disponemos de **herramientas que nos facilitan la creación, manipulación, búsqueda y ordenación de arrays**. Por ejemplo, para saber **cuántos elementos tiene un array**, escribiríamos:

```
Write-Host $a.Length
```

<https://docs.microsoft.com/en-us/dotnet/api/system.array?redirectedfrom=MSDN&view=net-6.0>

Recorrer un array

Muchas veces necesitamos **realizar operaciones que afecten a todos los elementos de un array**. Para estas situaciones podemos utilizar una estructura repetitiva controlada por una variable que actúe como índice del array. Un ejemplo sería cómo multiplicar por dos cada uno de los valores contenidos en un array.

```
$enteros = @(1, 2, 3, 4, 5, 6, 7, 8, 9)
for ($i = 0; $i -lt $enteros.Length; $i++) {
    $enteros[$i] = $enteros[$i] * 2
}
```

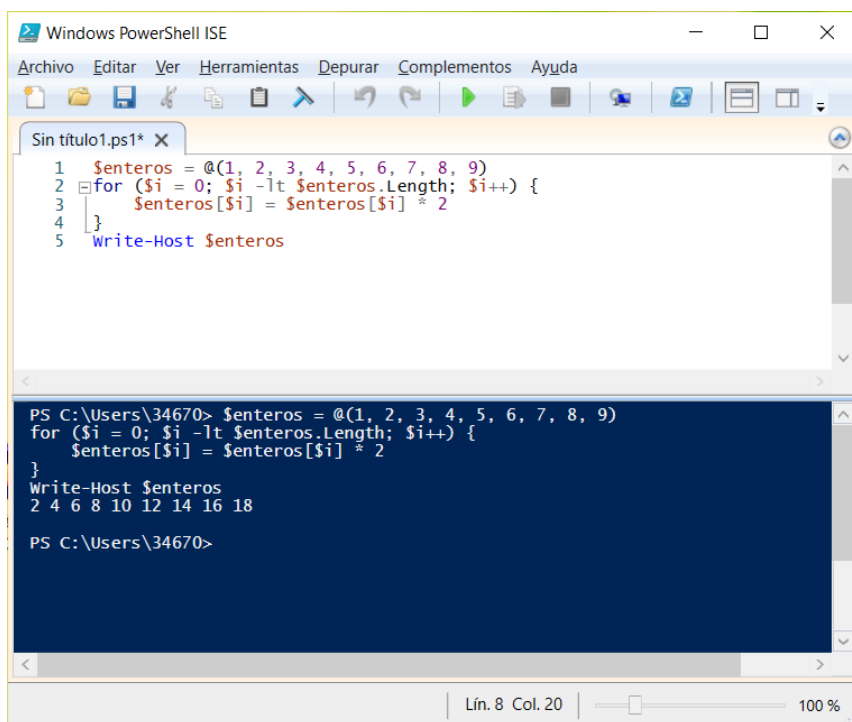
La **variable \$i comienza con el valor cero** y, en cada iteración, **se incrementa en uno** hasta llegar al valor de la **última posición del array**. Luego, dentro del bloque de código, se utiliza la **variable \$i para hacer referencia a un elemento particular del array** (primero al de la posición 0, luego al de la posición 1, etc.).

Si necesitáramos recorrer el array en orden inverso, bastaría con **modificar los valores de for**.

```
$enteros = @(1, 2, 3, 4, 5, 6, 7, 8, 9)
for ($i = $enteros.Length; $i -ge 0; $i--) {
    $enteros[$i] = $enteros[$i] * 2
}
```

Por supuesto, nos **vale cualquier estructura repetitiva**, siempre que actúe de forma equivalente a ésta. No obstante, existe también una estructura que está específicamente diseñada para actuar sobre colecciones de datos: la **estructura foreach**. Una de las ventajas de usar foreach para recorrer un array es que no necesitamos una variable que vaya tomando el valor de cada posición.

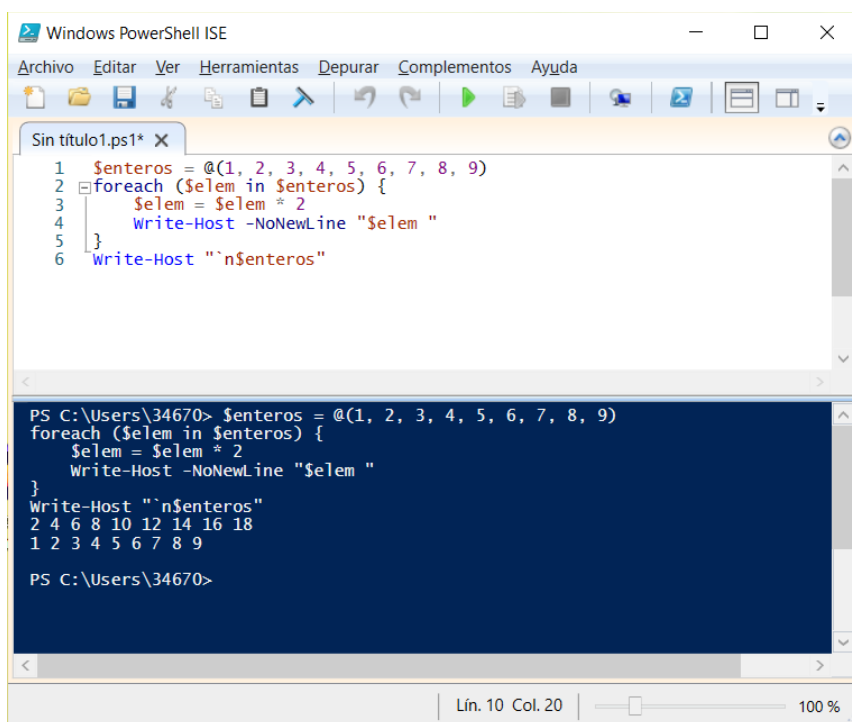
```
$enteros = @(1, 2, 3, 4, 5, 6, 7, 8, 9)
foreach ($elem in $enteros) {
    $elem = $elem * 2
}
```



```
Sin titulo1.ps1* X
1 $Senteros = @(1, 2, 3, 4, 5, 6, 7, 8, 9)
2 for ($i = 0; $i -lt $Senteros.Length; $i++) {
3     $Senteros[$i] = $Senteros[$i] * 2
4 }
5 Write-Host $Senteros

PS C:\Users\34670> $Senteros = @(1, 2, 3, 4, 5, 6, 7, 8, 9)
for ($i = 0; $i -lt $Senteros.Length; $i++) {
    $Senteros[$i] = $Senteros[$i] * 2
}
Write-Host $Senteros
2 4 6 8 10 12 14 16 18
PS C:\Users\34670>
```

De cualquier modo, debemos tener en cuenta una sutil diferencia: mientras **en el caso de for** estamos accediendo a un elemento del array cada vez, **en el caso de foreach** se hace una copia de dicho elemento en la variable `$elem`. Esto significa que, **al cambiar el valor de la variable `$elem`, no estamos cambiando el valor original en el array.**



```
Sin titulo1.ps1* X
1 $Senteros = @(1, 2, 3, 4, 5, 6, 7, 8, 9)
2 foreach ($elem in $Senteros) {
3     $elem = $elem * 2
4     Write-Host -NoNewLine "$elem "
5 }
6 Write-Host "`n$Senteros"

PS C:\Users\34670> $Senteros = @(1, 2, 3, 4, 5, 6, 7, 8, 9)
foreach ($elem in $Senteros) {
    $elem = $elem * 2
    Write-Host -NoNewLine "$elem "
}
Write-Host "`n$Senteros"
2 4 6 8 10 12 14 16 18
1 2 3 4 5 6 7 8 9
PS C:\Users\34670>
```

Hemos utilizado un **argumento** en el cmdlet `Write-Host`. Se trata de **`-NoNewLine`**, que evita que se produzca un cambio de línea después de ejecutarlo. Así, hemos conseguido que todos los números mostrados en el interior de `foreach` (cada uno de los `$elem`) aparezcan en la misma línea. También hemos utilizado un **carácter especial** (``n`) en el último `Write-Host` para, precisamente, **producir un cambio de línea.**

Otro inconveniente de **foreach** es **que siempre recorre todo el array**. Si necesitáramos recorrer sólo algunas de sus posiciones, o no quisiéramos hacerlo en el orden preestablecido, quizás sería más recomendable una estructura diferente.

Por ejemplo, imaginemos que tenemos un array que guarda nombres de productos en las posiciones impares y su precio correspondiente en las posiciones pares. Para aplicar un descuento del 10% a todos los productos, la solución más sencilla pasa por no utilizar foreach. Por ejemplo, podríamos aplicar la siguiente solución:

```
$productos = @(“Manzanas”, [float] 120,
               “Piña”, [float] 150,
               “Fresas”, [float] 30,
               “Melocotones”, [float] 14000)

for ($i = 0; $i -lt $productos.Length; $i++) {
    $productos[$i] = $productos[$i] * 0.9
}
```

Como PowerShell no tiene en cuenta los saltos de línea ni el exceso de espacios, podemos escribir un código más legible añadiéndolos según nuestro criterio.

Añadir y quitar valores en un array

Cuando necesitamos que un **array crezca con nuevos elementos**, basta con utilizar el **operador de suma (+)**. Por ejemplo, podemos crear un array vacío y añadir elementos a medida que los vaya facilitando un usuario:

```
$lista = @()

Write-Host “Introduce los nombres de la lista: ”
Write-Host “Para acabar, déjalo vacío y pulsa Intro.”

do {
    $nombre = Read-Host “Nuevo nombre”
    if ($nombre -ne "") { $lista = $lista + $nombre }
} while ($nombre)

Write-Host “Ésta es la lista introducida: ”
Write-Host $lista
```

También podemos usar el operador de suma para unir dos listas y obtener una tercera:

```
$a = @(1, 2, 3, 4)
$b = @(5, 6, 7, 8)
$c = $a + $b
```

Incluso podemos añadir una lista a otra:

```
$a = @(1, 2, 3, 4)
$b = @(5, 6, 7, 8)
$a += $b
```

En ocasiones, la **salida de un cmdlet consiste en un array**. Esto significa que **podríamos asignarla directamente a una variable de este tipo**. Un ejemplo puede ser el cmdlet **Get-ChildItem** que permite obtener la lista de archivos de una o varias rutas del disco. Para comprobar que su salida consiste en un array, bastaría con escribir lo siguiente:

```
Write-Host ((Get-ChildItem) -is [array])
```

Como el resultado es satisfactorio (\$true), podemos utilizar un array para almacenar la información que nos ofrece.

The screenshot shows the Windows PowerShell ISE interface. The script editor at the top contains the following code:

```

1 $lista = @()
2
3 Write-Host "Introduce los nombres de la lista: "
4 Write-Host "Para acabar, déjalo vacío y pulsa Intro."
5
6 do {
7     $nombre = Read-Host "Nuevo nombre"
8     if ($nombre -ne "") { $lista = $lista + $nombre }
9 } while ($nombre)
10
11 Write-Host "Esta es la lista introducida: "
12 Write-Host $lista

```

The console window at the bottom shows the execution of this script. The output is as follows:

```

PS C:\Users\34670> $lista = @()

Write-Host "Introduce los nombres de la lista: "
Write-Host "Para acabar, déjalo vacío y pulsa Intro."

do {
    $nombre = Read-Host "Nuevo nombre"
    if ($nombre -ne "") { $lista = $lista + $nombre }
} while ($nombre)

Write-Host "Esta es la lista introducida: "
Write-Host $lista
Introduce los nombres de la lista:
Para acabar, déjalo vacío y pulsa Intro.
Nuevo nombre: Mario
Nuevo nombre: Dylan
Nuevo nombre: Alejandro
Nuevo nombre: Cristóbal
Nuevo nombre:
Esta es la lista introducida:
Mario Dylan Alejandro Cristóbal

PS C:\Users\34670>

```

The status bar at the bottom indicates "Completado" (Completed) at line 23, column 20, with a zoom level of 100%.

```
$archivos = Get-ChildItem
```

En realidad, **Get-ChildItem** devuelve un array de objetos donde cada uno representará un archivo del **directorio**. La propiedad **name** de uno de esos objetos nos ofrecerá el nombre del archivo correspondiente. Sabiendo esto, podríamos utilizar una estructura repetitiva que recorra el array y nos muestre los nombres de los archivos contenidos en el directorio.

```
$archivos = Get-ChildItem
```

```
foreach ($elemento in $archivos) {
    Write-Host $elemento.name
}
```

En cuanto a la **eliminación de elementos de una array**, no hay un procedimiento sencillo. La mejor solución suele ser **crear un array nuevo y asignarle únicamente los valores que nos interese mantener**. Por ejemplo, en el array anterior, únicamente queremos mantener los nombres de los archivos que comiencen con la letra "D". Podríamos hacer algo así:

```

$parcial = @()
$archivos = Get-ChildItem

foreach ($elemento in $archivos) {
    if ($elemento.name.Substring(0,1) -eq "D") {
        $parcial = $parcial + $elemento
    }
}

$archivos = $parcial
$parcial = $null

```

The screenshot shows the Windows PowerShell ISE interface. The script editor at the top contains the following code:

```

1 $archivos = Get-ChildItem
2
3 foreach ($elemento in $archivos) {
4     Write-Host = $elemento.name
5 }

```

The console window at the bottom shows the execution of this script, resulting in the following output:

```

PS C:\Users\34670> $archivos = Get-ChildItem

foreach ($elemento in $archivos) {
    Write-Host = $elemento.name
}
= 3D objects
= Contacts
= Desktop
= Documents
= Downloads
= Ejemplo
= Favorites
= knime-workspace
= Links
= Music
= OneDrive
= Pictures
= Saved Games
= Searches
= Tracing
= Videos
= NULL

PS C:\Users\34670>

```

The status bar at the bottom indicates 'Completado' (Completed) and shows the cursor is at line 24, column 20.

Podemos simplificar el código utilizando el operador de concatenación de salida y entrada (tubería) de la siguiente forma, donde **\$_** hace referencia a cada uno de los elementos del array.

```

$parcial = @()
$archivos = Get-ChildItem

$archivos = $archivos | where { $_.name.Substring(0,1) -eq "D" }

```

Tablas hash

Las tablas Hash son **estructuras de datos que almacenan parejas de claves y valores**. También se conocen como **diccionarios o arrays asociativos**.

Se trata de **arrays donde, en lugar de identificar los diferentes elementos por su posición, utilizan valores que actúan como índices**. Su mayor ventaja es que **las claves y los valores pueden ser de cualquier tipo y longitud**. Su sintaxis es parecida a la de los arrays, pero cambiando los paréntesis por llaves y las comas por puntos y comas:

```
$variable = @(clave1 = valor1, clave2 = valor2, clave3 = valor3)
```

Veamos un array reinterpretado como tabla hash:

```

$productos = @{"Manzanas" = [float] 120;
               "Piña" = [float] 150;
               "Fresas" = [float] 30;
               "Melocotones" = [float] 14000}

```

Incluso podemos **crear una tabla hash vacía e ir añadiendo elementos** más tarde:

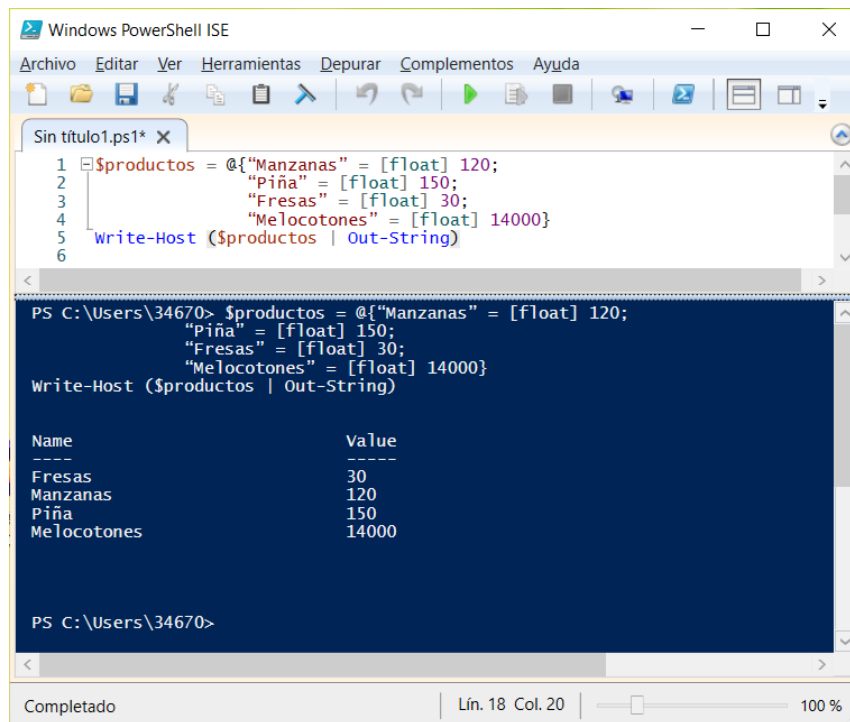
```
$productos = @{}
```

Mostrar el contenido de una tabla hash

A diferencia de lo que ocurriría con los arrays, para que Write-Host consiga mostrar el contenido de una tabla hash, **debemos canalizar su contenido a través del cmdlet Out-String**. Éste **convierte la información ofrecida por los objetos de PowerShell en arrays**. Por lo tanto, la sintaxis que usaremos sería:

```
Write-Host ($productos | Out-String)
```

El resultado será como el que ilustra la imagen siguiente:



También podemos consultar **sólo sus claves utilizando la propiedad Keys** o **sólo los valores usando la propiedad Values**:

```
Write-Host $productos.Keys  
Write-Host $productos.Values
```

Utilizar un elemento individual de la tabla hash

Para **acceder a un elemento concreto de la tabla hash, basta con escribir su nombre seguido de la clave que necesitamos encerrada entre corchetes**. Por ejemplo, para mostrar el primer elemento de la tabla hash anterior, escribiríamos esto:

```
Write-Host $productos["Manzanas"]
```

También podemos utilizar **notación orientada a objetos** y el resultado que obtenemos es equivalente al de la notación convencional. Incluso podemos prescindir de las comillas si la clave consiste en una sola palabra.

```
Write-Host $productos."Manzanas"  
Write-Host $productos.Manzanas
```

Y si lo que queremos es **utilizar el elemento dentro de una expresión**, nos limitamos a utilizar la notación anterior como si se tratara de una variable simple. Por ejemplo, para incrementar en uno el valor del elemento.

```
$productos["Manzanas"] = $productos["Manzanas"] + 1  
$productos["Manzanas"]++
```

Tabla hash como objeto

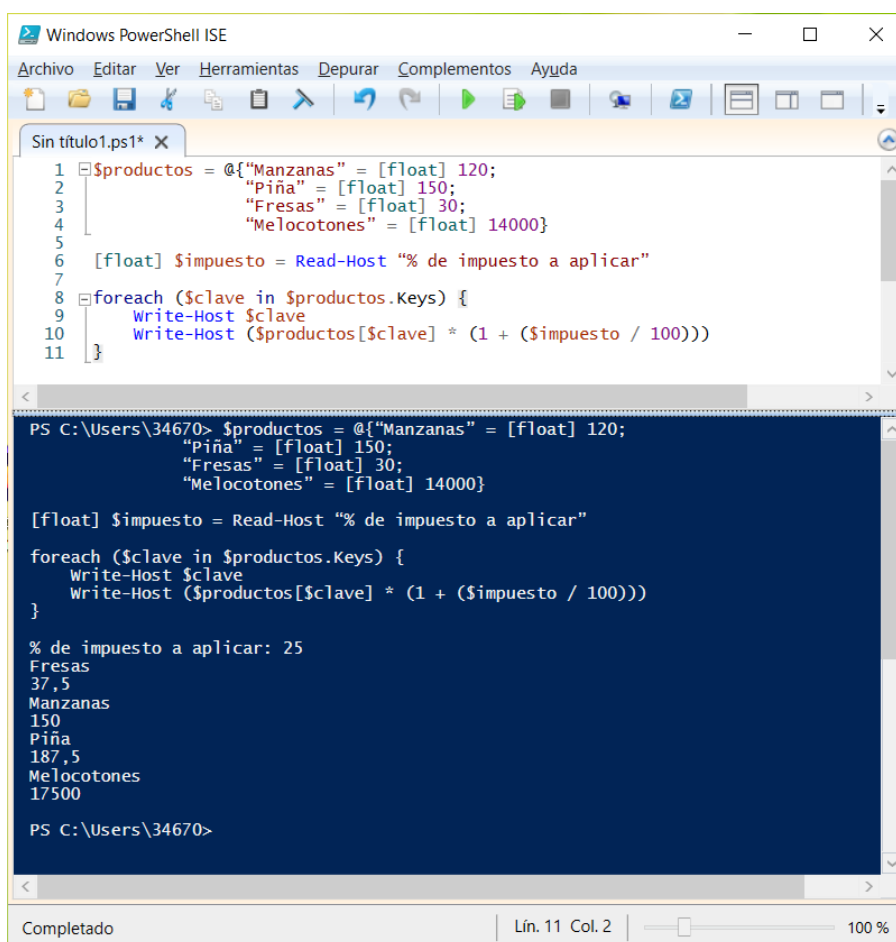
Como en el caso de los arrays, en las tablas hash, **disponemos de herramientas que nos facilitan su creación, manipulación, búsqueda y ordenación**. Por ejemplo, para obtener el número de pares contenidos en una tabla hash escribiríamos un código similar al siguiente:

```
Write-Host $productos.Count
```

<https://docs.microsoft.com/es-es/dotnet/api/system.collections.hashtable?redirectedfrom=MSDN&view=net-6.0>

Recorrer una tabla hash

Muchas veces necesitamos **realizar operaciones que afecten a todos los elementos de la tabla hash**. Para estas situaciones **podemos utilizar la estructura repetitiva foreach**. Por ejemplo, imagina que quieres obtener el precio de los artículos anteriores, después de añadirles un impuesto.



The screenshot shows the Windows PowerShell ISE interface. The script in the editor is as follows:

```
1 $productos = @{"Manzanas" = [float] 120;  
2   "Piña" = [float] 150;  
3   "Fresas" = [float] 30;  
4   "Melocotones" = [float] 14000}  
5  
6 [float] $impuesto = Read-Host "% de impuesto a aplicar"  
7  
8 foreach ($clave in $productos.Keys) {  
9   Write-Host $clave  
10  Write-Host ($productos[$clave] * (1 + ($impuesto / 100)))  
11 }
```

The console output shows the execution of the script:

```
PS C:\Users\34670> $productos = @{"Manzanas" = [float] 120;  
"Piña" = [float] 150;  
"Fresas" = [float] 30;  
"Melocotones" = [float] 14000}  
  
[float] $impuesto = Read-Host "% de impuesto a aplicar"  
  
foreach ($clave in $productos.Keys) {  
  Write-Host $clave  
  Write-Host ($productos[$clave] * (1 + ($impuesto / 100)))  
}  
  
% de impuesto a aplicar: 25  
Fresas  
37,5  
Manzanas  
150  
Piña  
187,5  
Melocotones  
17500  
  
PS C:\Users\34670>
```

```
$productos = @{"Manzanas" = [float] 120;  
  "Piña" = [float] 150;  
  "Fresas" = [float] 30;  
  "Melocotones" = [float] 14000}  
  
[float] $impuesto = Read-Host "% de impuesto a aplicar"
```

```
foreach ($clave in $productos.Keys) {  
    Write-Host $clave  
    Write-Host ($productos[$clave] * (1 + ($impuesto / 100)))  
}
```

Añadir y quitar elementos en una tabla hash

Cuando necesitamos que una **tabla hash crezca con nuevos elementos**, basta con escribir su nombre **seguido de la nueva clave encerrada entre corchetes**:

```
$productos["Limones"] = [float] 200
```

También podemos utilizar sintaxis orientado a objetos:

```
$productos.Add("Limones", [float] 200)  
$productos.Limones = [float] 200
```

Y para **eliminar uno de los elementos**, bastaría con escribir:

```
$productos.Remove("Limones")
```

Por último, si necesitamos **eliminar la tabla completa**, podríamos utilizar su método Clear:

```
$productos.Clear()
```