

PowerShell

Punto 1: ¿Qué es PowerShell? -----	3
Introducción -----	3
Ejecutar PowerShell -----	4
Comandos y cmdlets en PowerShell -----	7
Obtener más ayuda -----	11
Redirecciones y tuberías -----	14
Redirecciones -----	14
Tuberías -----	15
Punto 2: Los primeros pasos -----	18
Antes de empezar -----	18
Escribir el primer script -----	19
Ejecutar el primer script -----	21
El entorno de desarrollo integrado Windows PowerShell ISE -----	23
Crear comandos de forma interactiva en PowerShell ISE -----	27
Crear un nuevo script en PowerShell ISE -----	30
Guardar un script en PowerShell ISE para usarlo en el futuro. -----	33
Abrir en PowerShell ISE un script que ya existe -----	34
Punto 3: Variables en PowerShell -----	35
Concepto de variable -----	35
Usar variables en PowerShell -----	35
Pedir información al usuario -----	37
Mostrar información al usuario -----	38
Tipo de dato de una variable -----	38
Definición implícita: -----	41
Definición explícita: -----	42
Especificar el tipo de un dato -----	42
Espacio ocupado por las variables en memoria -----	45
Punto 4: Operaciones básicas con variables -----	45
Introducción -----	45
Operar con variables de tipo texto -----	46
Operar con variables numéricas -----	47
Operadores aritméticos especiales -----	49
Operar con variables lógicas -----	50
Operadores lógicos -----	53

Operadores de tipo-----	55
Variables como objetos -----	58
Punto 5: Estructuras condicionales -----	61
Introducción -----	61
Uso de la orden if -----	61
Uso de la orden if con else -----	63
Uso de la orden if con elseif -----	64
Uso de la orden switch -----	66
Punto 6: Estructuras repetitivas -----	74
Introducción -----	74
La estructura do while -----	74
Como hemos dicho más arriba, la estructura do while nos permite repetir un	74
La estructura while -----	76
La estructura do until -----	77
La estructura for -----	78
La estructura foreach -----	79
Alterar el proceso de las estructuras repetitivas -----	83
La orden break -----	83
La orden continue -----	84
Punto 7: Variables que guardan múltiples valores -----	85
Introducción -----	85
Arrays -----	85
Crear un array -----	85
Mostrar el contenido de un array -----	86
Utilizar un elemento individual del array -----	87
Arrays como objetos -----	88
Añadir y quitar valores en un array -----	91
Tablas Hash -----	96
Mostrar el contenido de una tabla Hash -----	97
Utilizar un elemento individual de la tabla Hash -----	97
Tabla Hash como objetos -----	99
Recorrer una tabla Hash -----	100
Añadir y quitar elementos en una tabla Hash -----	101

Punto 1: ¿Qué es PowerShell?

Introducción

Desde los principios 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 la línea de comandos era crear archivos de procesos por lotes. Un mecanismo bastante rudimentario y muy limitado que ha ido mostrando sus carencias con el paso del tiempo.

Afortunadamente, en abril de 2006 *Microsoft* lanzó una nueva interfaz de consola, con un nutrido grupo de órdenes y la capacidad de crear *scripts* usando una sintaxis moderna, que comparte similitudes con el lenguaje *Perl*. Esta nueva interfaz comenzó denominándose *Monad* durante su periodo de desarrollo, pero terminó llamándose *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 *MAC OSX*.

Las ordenes incluidas en *PowerShell* son muchas, y reciben el nombre de *cmdlets* (de *command-let*). Al principio, *PowerShell* tenía sólo 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.

Versiones de PowerShell	
Versión	Año
v1	2006
v2	2009
v3	2012
v4	2013
v5	2015

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.

Dicho así, puede que asuste un poco, pero aquí vamos a mostrarte las

bases que te permitan crear tus primeros *scripts* de una forma sencilla y, como siempre, con multitud de ejemplos que te guiarán a lo largo del camino.

Durante nuestra explicación, utilizaremos *Windows 10*, pero no debes olvidar que *PowerShell* está disponible en todos los sistemas operativos de *Microsoft* desde *Windows 7* y nuestras explicaciones serán válidas en todos ellos.

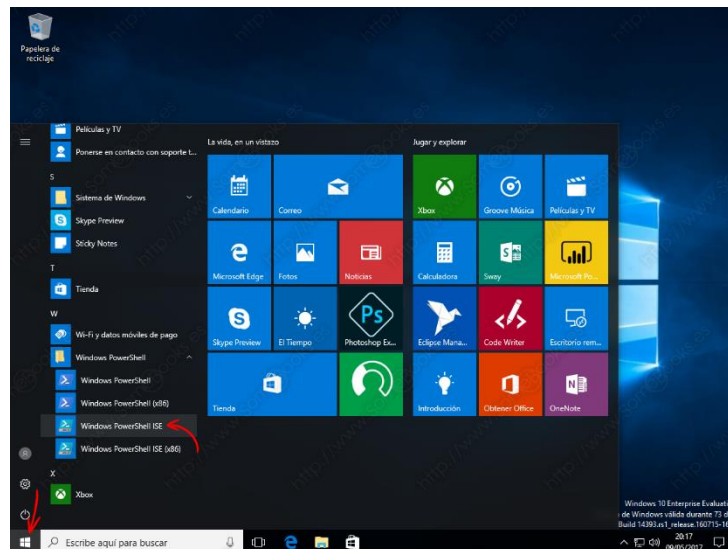
Ejecutar PowerShell

Tenemos dos opciones a la hora de ejecutar *PowerShell*:

1. Una opción con entorno gráfico llamada *PowerShell ISE* (del inglés, *Integrated Scripting Environment*).

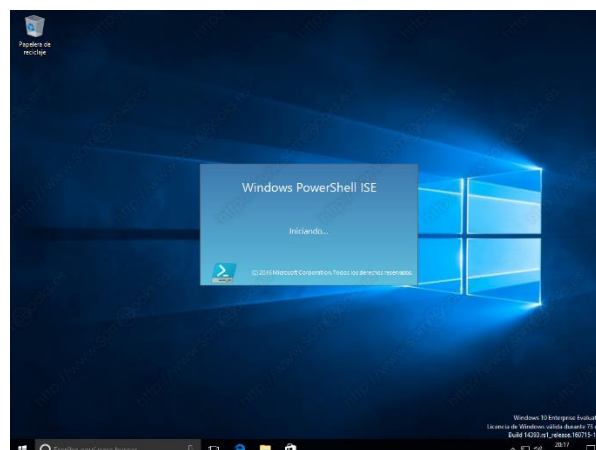
Para utilizarla, basta con hacer clic sobre el botón *Inicio* y desplazarnos por el menú hasta la carpeta *Windows PowerShell*.

En su interior, hacemos clic sobre el elemento *Windows PowerShell ISE*.



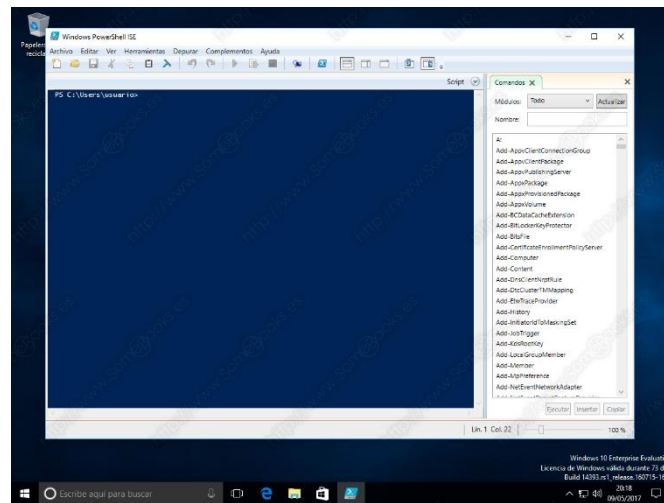
Observa que también tenemos una opción titulada *Windows PowerShell ISE (x86)*. Esta es la versión que deberás elegir si tu equipo tiene una arquitectura de 32 bits.

Durante unos instantes veremos la pantalla de carga de *Windows PowerShell ISE*.



Poco después se abrirá el entorno de desarrollo de PowerShell.

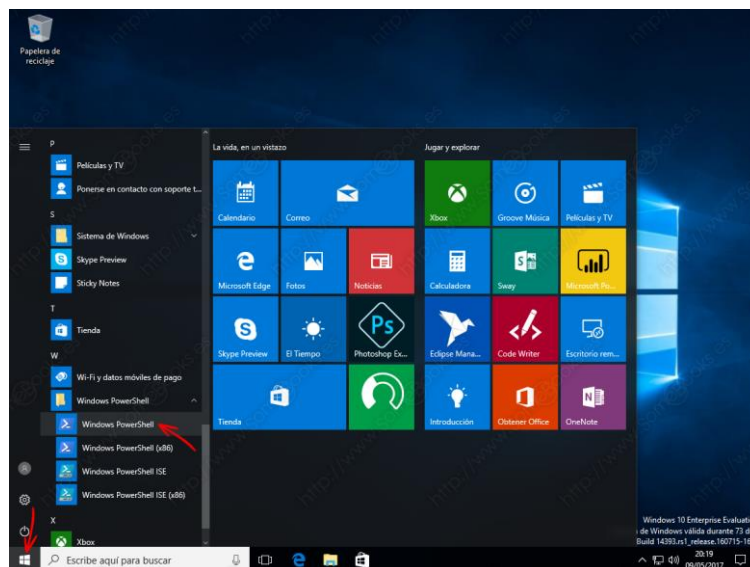
A partir de este momento, podremos comenzar a trabajar con las diferentes opciones de la interfaz gráfica.



2. Una consola de texto, que se parece a la línea de comandos tradicional.

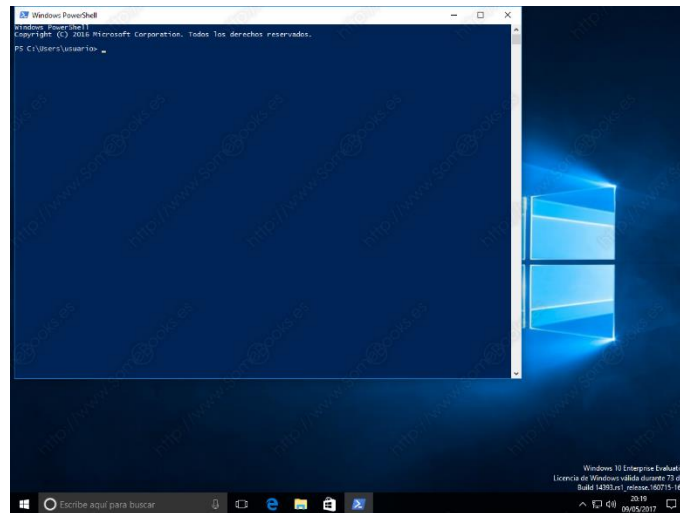
Como antes, para utilizarla comenzaremos por hacer clic sobre el botón *Inicio* y desplazarnos por el menú hasta la carpeta *Windows PowerShell*.

Una vez allí, haremos clic sobre el elemento *Windows PowerShell*.



También en este caso disponemos de una opción titulada *Windows PowerShell (x86)* para equipos con arquitectura de 32 bits.

Al poco aparecerá una ventana, lista para comenzar a escribir nuestras órdenes.

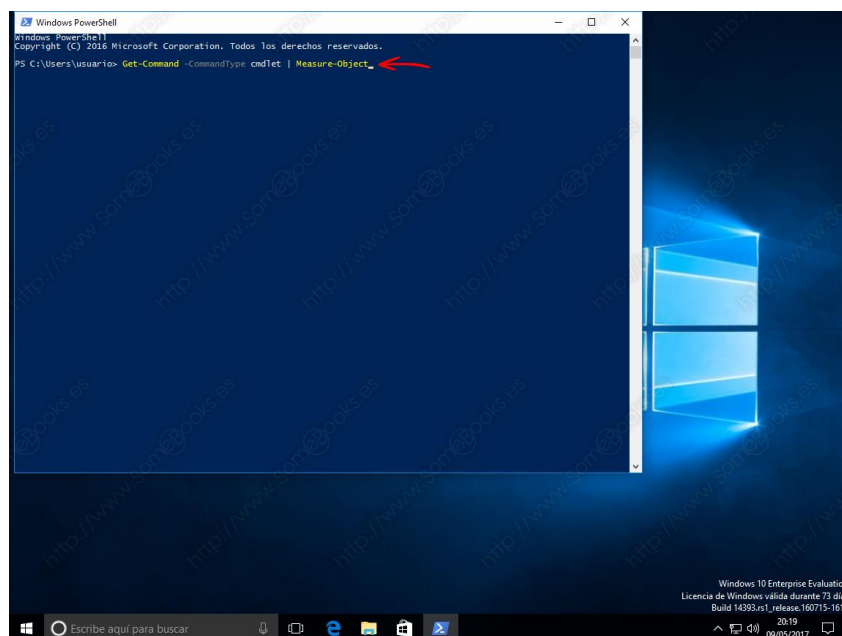


Para nuestros primeros ejemplos, usaremos la *consola de texto*, que puede ser más sencilla al principio. Después, cuando comencemos a escribir *scripts*, nos pasaremos a *PowerShell ISE*.

Si estás deseando comprobar cómo funciona, puedes aprovechar para conocer el número exacto de *cmdlets* que tienes disponibles en tu sistema. Para lograrlo, basta con ejecutar la siguiente orden:

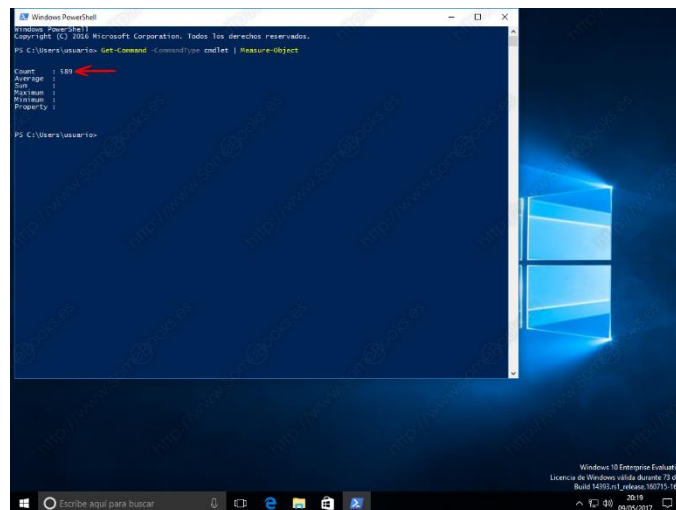
```
Get-Command -CommandType cmdlet | Measure-Object
```

Escribimos el comando en la consola y pulsamos la tecla Intro.



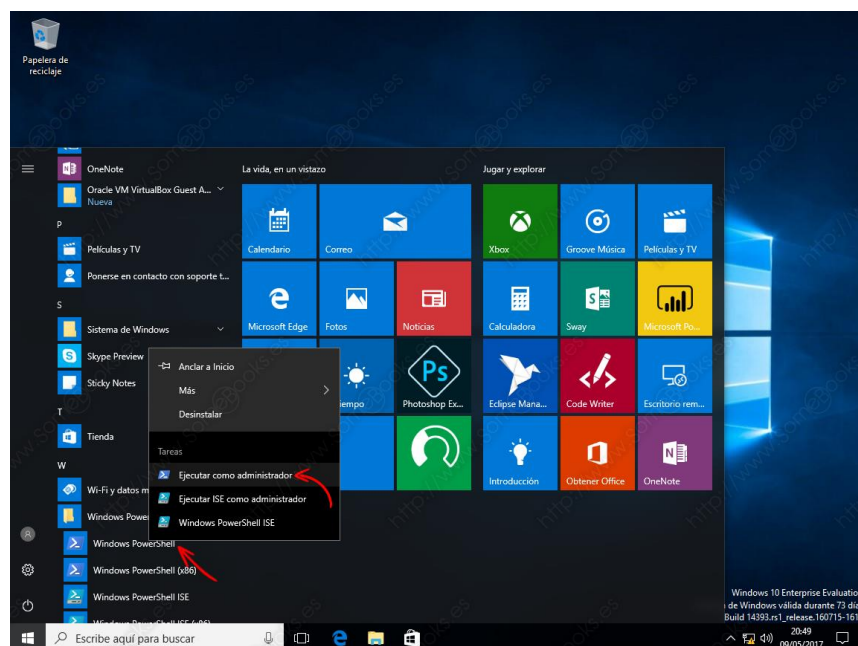
Al momento, tendremos la respuesta en la pantalla.

En mi caso, aparecen 589, pero pueden ser muchos más a medida que instalemos software complementario



Por último, si necesitas ejecutar *PowerShell* con privilegios administrativos, bastará con hacer clic, sobre el elemento *Windows PowerShell*, con el botón derecho del ratón.

Y en el menú de contexto que aparece, elegir la opción *Ejecutar como administrador*.



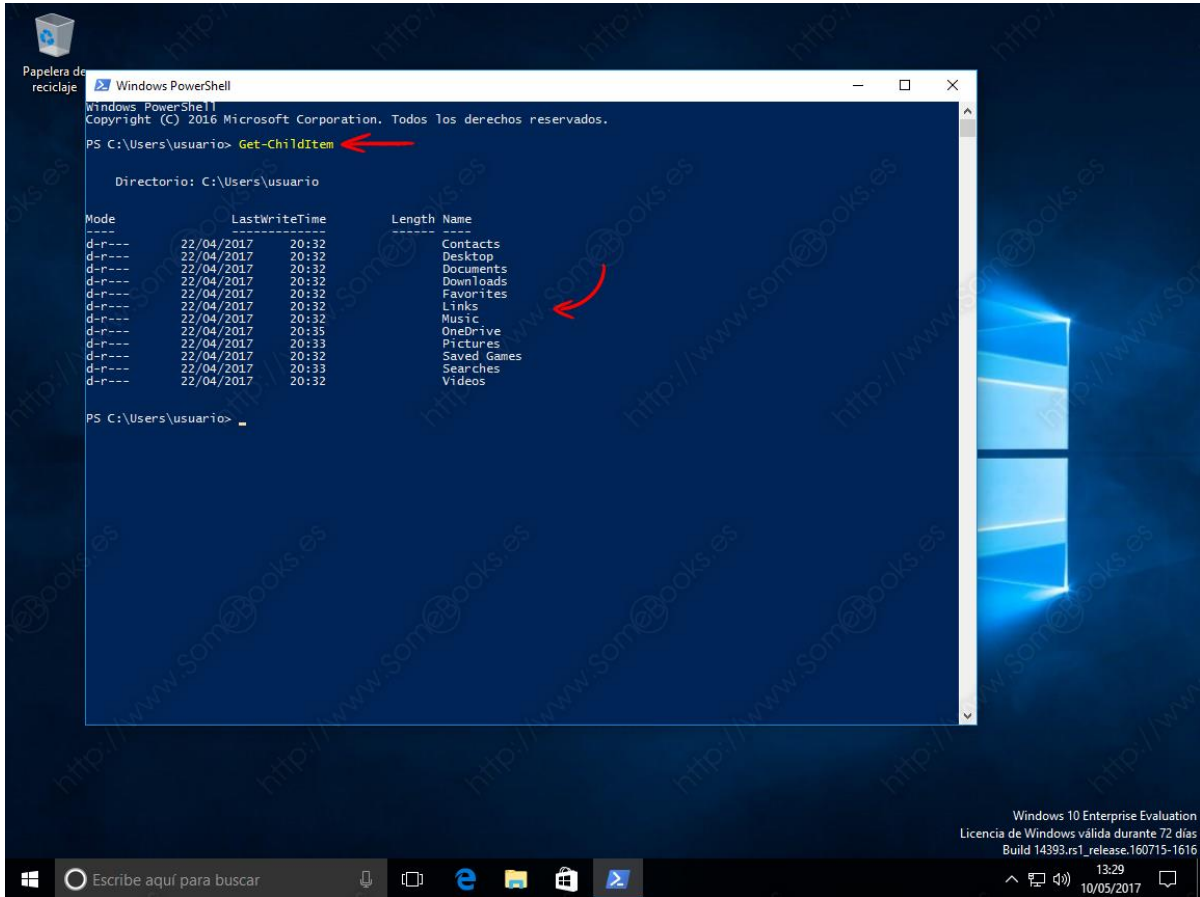
Comandos y cmdlets en PowerShell

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:

Get-ChildItem

Salida del *cmdlet* Get-ChildItem.



```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. Todos los derechos reservados.

PS C:\Users\usuario> Get-ChildItem

Directorio: C:\Users\usuario

Mode                LastWriteTime         Length Name
----                -
d-r-----      22/04/2017    20:32             Contacts
d-r-----      22/04/2017    20:32             Desktop
d-r-----      22/04/2017    20:32             Documents
d-r-----      22/04/2017    20:32             Downloads
d-r-----      22/04/2017    20:32             Favorites
d-r-----      22/04/2017    20:32             Links
d-r-----      22/04/2017    20:32             Music
d-r-----      22/04/2017    20:35             OneDrive
d-r-----      22/04/2017    20:33             Pictures
d-r-----      22/04/2017    20:32             Saved Games
d-r-----      22/04/2017    20:33             Searches
d-r-----      22/04/2017    20:32             Videos

PS C:\Users\usuario>
```

Sin embargo, seguimos teniendo a nuestra disposición el comando `dir`, que hacía las mismas funciones en la antigua línea de comandos. Así, sólo escribiendo la orden, obtendré un resultado equivalente al anterior:

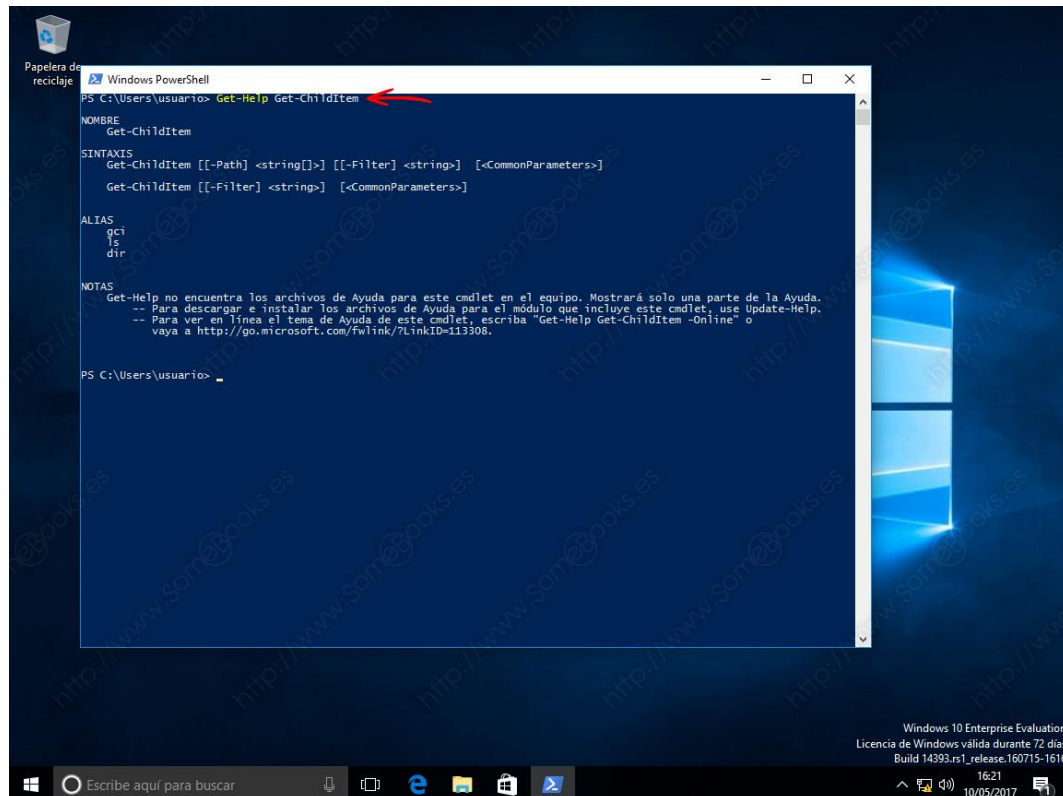
`dir`

Lo que hemos hecho con el comando `dir` es sólo una prueba, pero existe un gran número de comandos antiguos que siguen funcionando perfectamente en *PowerShell*. Esto significa que podrás comenzar a utilizar *PowerShell* con tus conocimientos anteriores, en lugar de aprender al principio un número elevado de *cmdlets*.

Puedes encontrar información sobre los *cmdlets* más comunes en <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.management/microsoft.powershell.management>.

También puedes obtener información sobre cualquier comando usando la ayuda del propio *PowerShell*. Para lograrlo, sólo tienes que utilizar el *cmdlet* **Get-Help**. Por ejemplo:

Get-Help Get-ChildItem



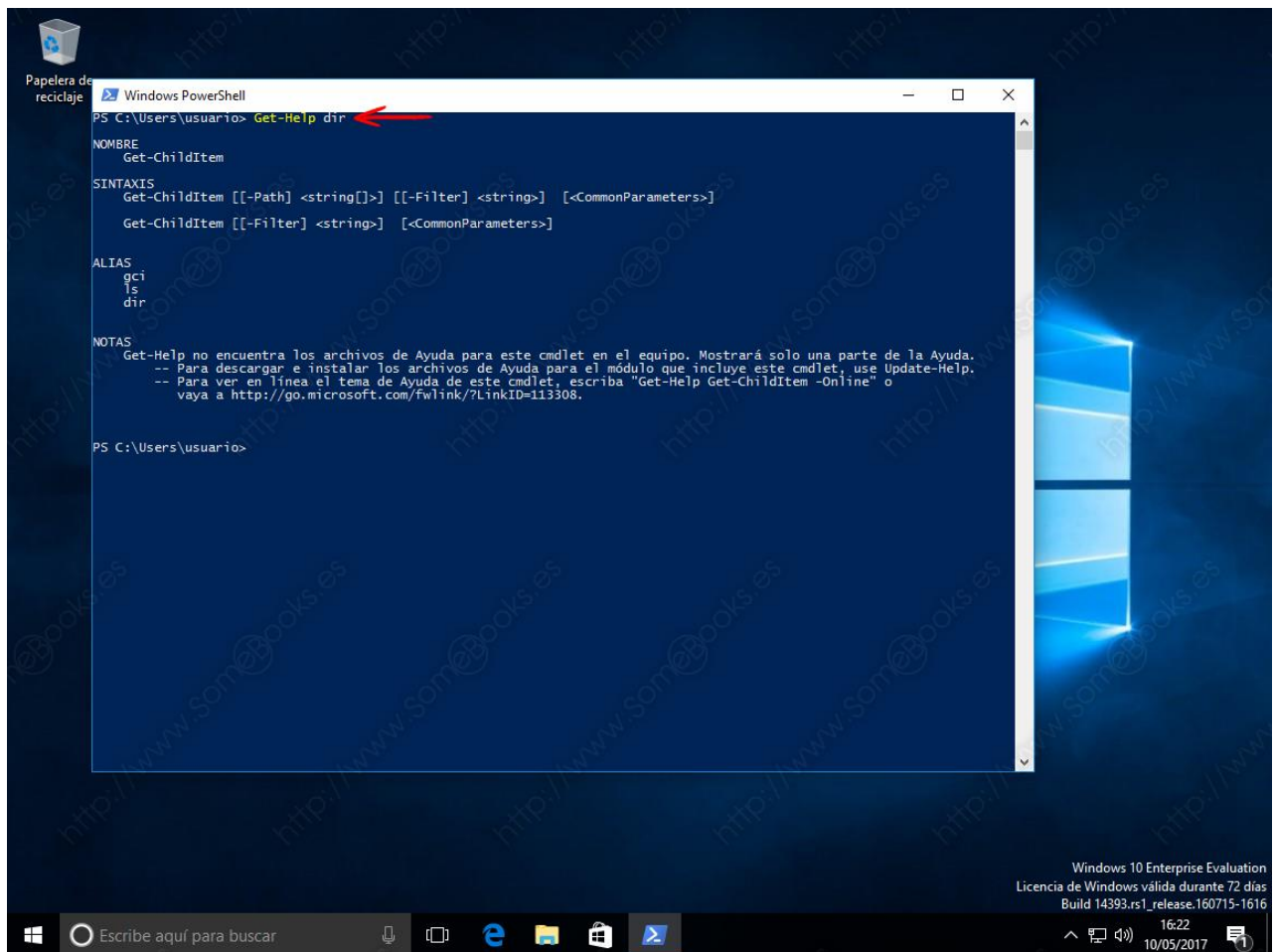
En realidad, la forma que tiene *PowerShell* de dar soporte a comandos antiguos es utilizando *alias*.

Podríamos decir que un *alias* es un sobrenombre, un *apodo*, 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 te fijas en la imagen anterior, la información que ofrece **Get-Help** sobre **Get-ChildItem** ya nos informa de que tiene tres *alias*: **gci**, **ls**, **dir**.

Esta característica puedes utilizarla también cuando conoces el nombre del *alias* (el comando antiguo), pero no conoces el *cmdlet* correspondiente. En ese caso, basta con utilizar **Get-Help** seguido del nombre del alias:

Get-Help dir

Como ves en la imagen, la ayuda ofrecida es la del *cmdlet* que se corresponde con el alias.

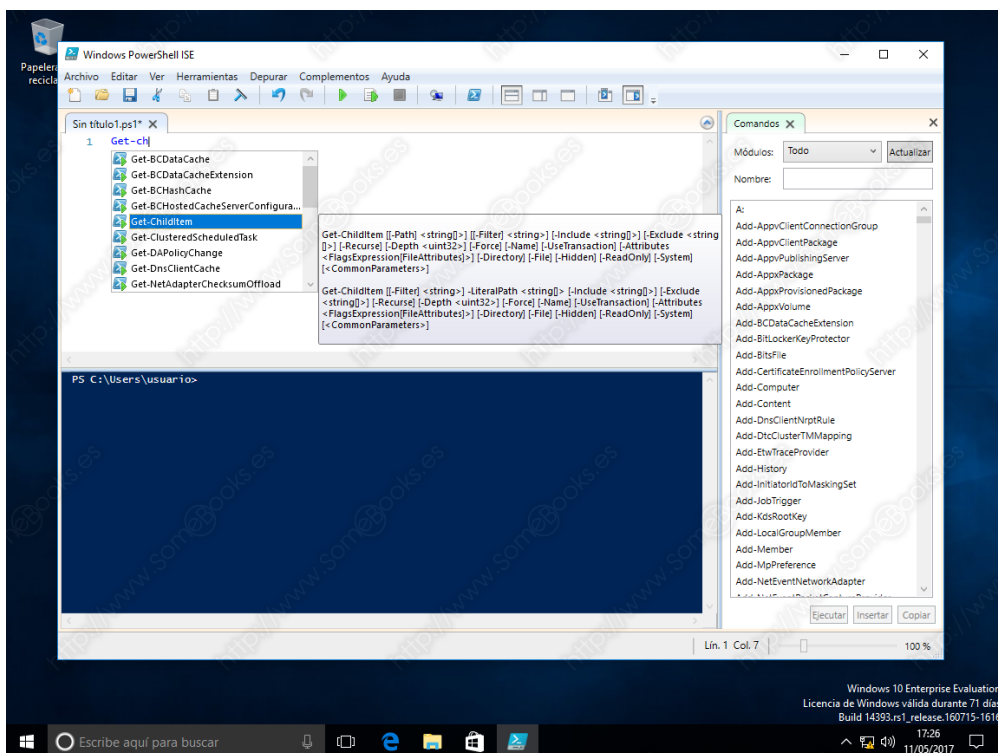


Los nombres de todos los **cmdlets** siguen un patrón común, formado por un verbo, un guión y un nombre en singular. De este modo será mucho más fácil recordarlos.

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.

Además, **PowerShell** dispone 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, bastará con pulsar la tecla `Intro`.



Obtener más ayuda

Si te fijas 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*:

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

Escribimos la orden y pulsamos la tecla `Intro`.

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* que estamos consultando.

1. Utilizar directamente en el navegador web la URL que nos ofrece **Get-Help** (para nuestro ejemplo, <http://go.microsoft.com/fwlink/?LinkID=113308>, aunque variará en función del *cmdlet* consultado).

Esto nos llevará exactamente a la misma página web de la imagen anterior.

2. Instalar los archivos de ayuda de forma local. 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.

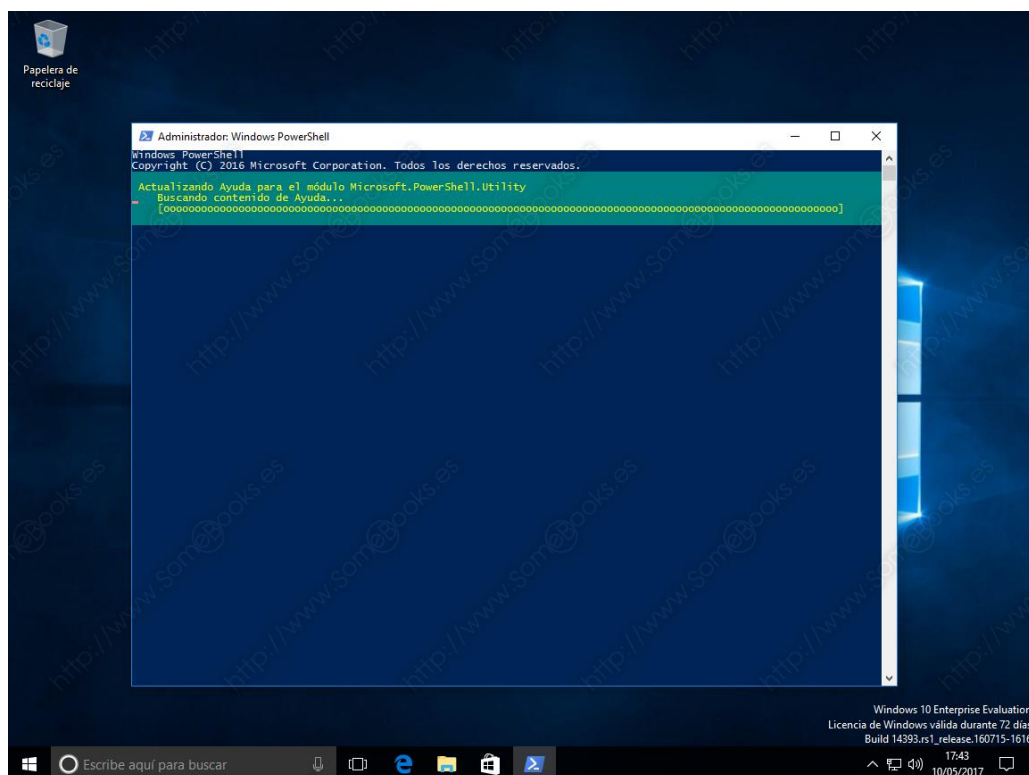
Por ese motivo, nosotros aquí instalaremos únicamente los módulos relacionados con **PowerShell**. Para lograrlo, sólo tenemos que incluir el argumento **-Module** seguido del nombre de los módulos de ayuda a instalar (en nuestro caso, todos los relacionados con **PowerShell**):

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

Como de costumbre, escribimos la orden y pulsamos la tecla Intro.

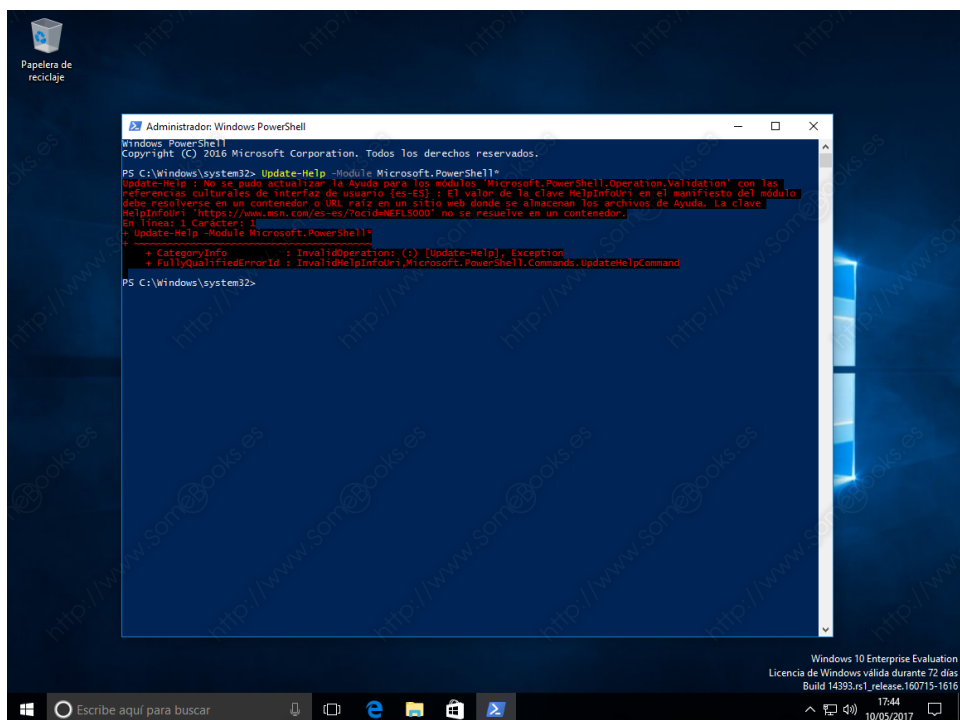
Al hacerlo, el sistema parará unos minutos descargando e instalando los archivos de ayuda de diferentes módulos

Sólo tenemos que esperar unos instantes



Puede que al final de la instalación, el sistema te avise de que algunos de los módulos de ayuda no se ha instalado satisfactoriamente. No debes preocuparte en exceso porque suelen ser módulos poco utilizados.

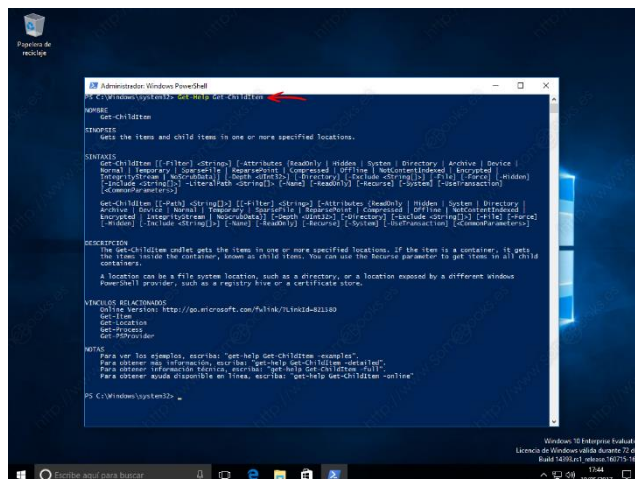
En mi caso, no se ha podido instalar el módulo **Microsoft.PowerShell.Operation.Validation**.



Si ahora volvemos a utilizar la sintaxis original de **Get-Help**, obtendremos una salida mucho más detallada que al principio:

Get-Help Get-ChildItem

Nueva salida de **Get-ChildItem**.



Como antes, en la parte final de la salida, nos recomiendan otras alternativas. Por ejemplo:

Get-Help Get-ChildItem -Detailed

Así, obtenemos una información aún más extendida

... O la opción:

```
Get-Help Get-ChildItem -Examples
```

... que nos ofrece ejemplos de uso del *cmdlet*.

Incluso podemos obtener información detallada de cada argumento usando el argumento **-Full**:

```
Get-Help Get-ChildItem -Full
```

En este caso, la salida es extensa y detallada.

Redirecciones y tuberías

Básicamente, tanto las *redirecciones* como las *tuberías* tienen como objetivo obtener la salida de una acción y dirigirla a un lugar diferente del predeterminado (que normalmente es la pantalla)

Redirecciones

Las redirecciones 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 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.

Veamos un ejemplo:

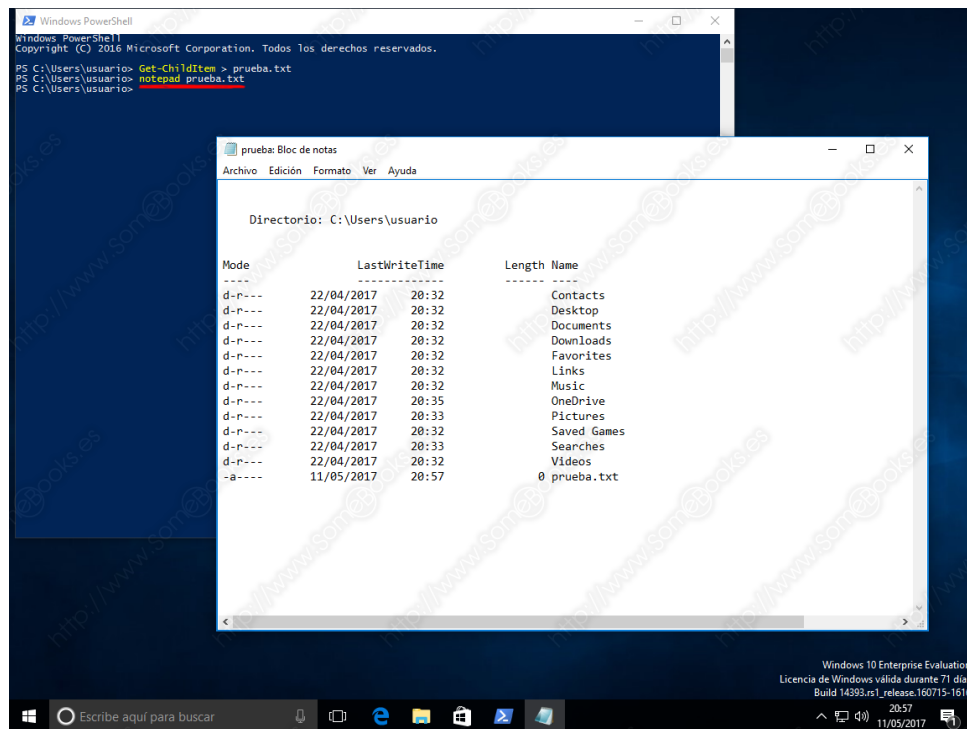
```
Get-ChildItem > prueba.txt
```

Escribimos la orden y pulsamos la tecla `Intro`.



A continuación, comprobamos el contenido del archivo

... Por ejemplo, utilizando el *bloc de notas*.



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

En este caso, el ejemplo quedaría así:

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

Volvemos a escribir la orden y pulsar la tecla **Intro**.

Y comprobamos de nuevo el contenido del archivo.

Donde comprobamos que, efectivamente, la nueva salida se ha añadido a la anterior.

Tuberías

En la mayoría de los teclados, para obtener el carácter pleca (**|**), usaremos la combinación de teclas **Alt Gr + 1**.

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 **pleca** (**|**), que suele llamarse **operador de canalización**. Por último, escribiremos el segundo **cmdlet**.

De hecho, el primer ejemplo que pusimos en esta documentación, ya usaba una tubería:

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

En este 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.

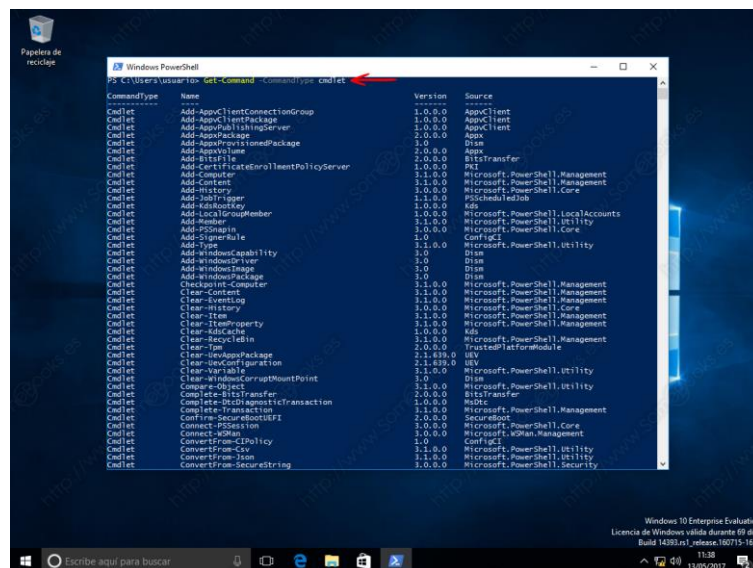
Para reducir el abanico, empleamos el parámetro **-CommandType**, donde le indicamos que sólo estamos interesados en los *cmdlets*.

Hasta aquí, estaríamos ejecutando un *cmdlet* individual, como habíamos hecho hasta ahora con **Get-Help** o **Get-ChildItem**:

```
Get-Command -CommandType cmdlet
```

Al ejecutarlo, tendríamos una relación completa de los *cmdlets* que tenemos disponibles en nuestro equipo.

La lista es mucho más larga de lo que se ve en la imagen..



Sin embargo, a pesar de que **Get-Command** puede ser muy útil, no era lo que necesitábamos en el momento de incluir por primera vez el ejemplo. Si recuerdas, el objetivo era saber cuántos *cmdlets* teníamos, no cómo se llamaban todos ellos.

Por ese motivo, redireccionamos la salida de **Get-Command** para que se utilizara 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 589.

Puedes encontrar información detallada sobre el *cmdlet* **Measure-Object**, en la dirección <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/measure-object>.

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 puedes combinar las tuberías con las redirecciones, obteniendo el resultado en un archivo. Por ejemplo, imagina que necesitas saber, en las carpetas de los usuarios del sistema, qué archivos exceden un determinado tamaño.

Primero debemos pensar en cómo obtener todos los archivos que hay en la carpeta *Users*. Pero esto ya lo sabemos hacer, basta con utilizar el *cmdlet* **Get-ChildItem** indicando la ruta donde buscamos y, en este caso, el parámetro **-Recurse** para que también busque en las subcarpetas:

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

Lógicamente, para tener acceso a los archivos de todos los usuarios, deberemos estar ejecutando *PowerShell* con privilegios administrativos.

El resultado será una lista de todos los archivos que hay en la ruta indicada. Para filtrar los que nosotros necesitamos, basta con enviar la salida al *cmdlet* **Where-Object**, que nos permite filtrar la salida anterior en función del tamaño del objeto (en nuestro ejemplo, los archivos mayores que 1 MB, pero tú puedes usar la cantidad que prefieras):

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

Finalmente, enviaremos la salida a un archivo para poder consultarla más tarde:

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

Punto 2: Los primeros pasos

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, basta con ejecutar el siguiente *cmdlet*:

```
Get-ExecutionPolicy
```

En nuestro caso, comprobamos que se encuentra habilitada la opción más restrictiva.

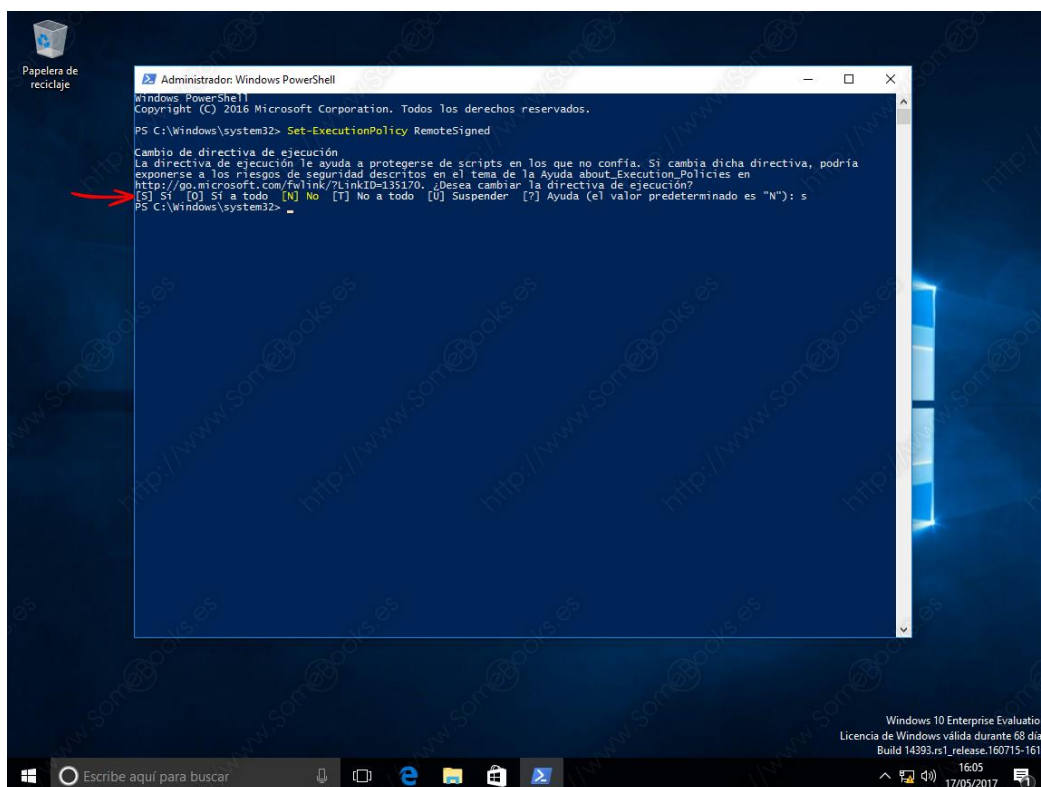
Para establecer la política de ejecución de scripts, necesitaremos ejecutar el *cmdlet* Set-ExecutionPolicy seguido del nombre de política elegido. Por ejemplo, para elegir RemoteSigned escribiremos lo siguiente:

```
Set-ExecutionPolicy RemoteSigned
```

Para ejecutar **Set-ExecutionPolicy** necesitamos privilegios administrativos. Recuerda que ya hemos hablado de cómo ejecutar *PowerShell* como administrador en el apartado *Ejecutar PowerShell* del punto 1.

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.

Para poder ejecutar los *scripts* que escribamos más adelante, ahora deberemos contestar pulsando la tecla `[S]`.



Puedes encontrar información detallada sobre los *cmdlets* **Get-ExecutionPolicy** y **Set-ExecutionPolicy** en el sitio web de Microsoft:

- **Get-ExecutionPolicy**: <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.security/get-executionpolicy>.
- **Set-ExecutionPolicy**: <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.security/set-executionpolicy>.

Escribir el primer script

Un *script* de *PowerShell* sólo es un archivo de texto plano que contiene la

secuencia de ordenes 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*.

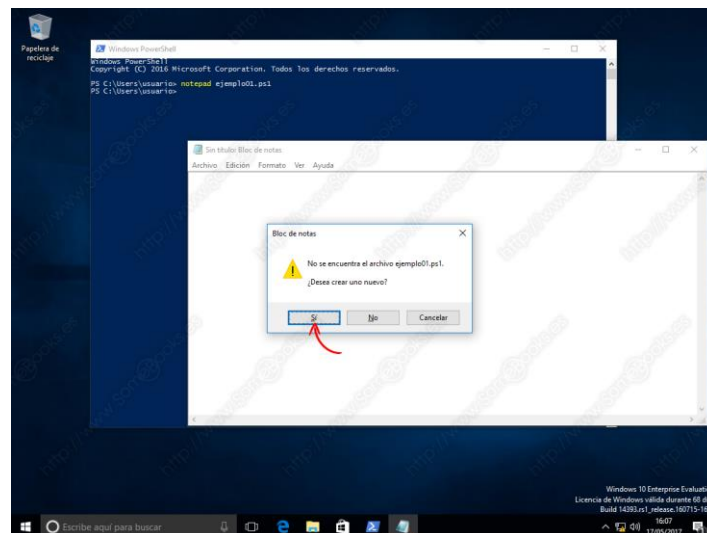
Por ejemplo, para crear nuestro primer *script* usaremos el editor de textos predeterminado de *Windows*. Es decir, el *Bloc de notas*. Por lo tanto, desde la línea de comandos de *PowerShell* escribiremos lo siguiente:

```
notepad ejemplo01.ps1
```

Como ves, ejecutamos el programa *notepad.exe* (que es el programa del *Bloc de notas*), enviándole como argumento el nombre del archivo que queremos editar. En nuestro caso, *ejemplo01.ps1*.

El efecto inmediato es que se abrirá el *Bloc de notas*. Si el archivo *ejemplo01.ps1* ya existiera, aparecería su contenido en el área de trabajo del editor. Sin embargo, como aún no existe, nos aparece un mensaje indicándonos si queremos crearlo.

Para continuar, hacemos clic sobre el botón *Sí*.



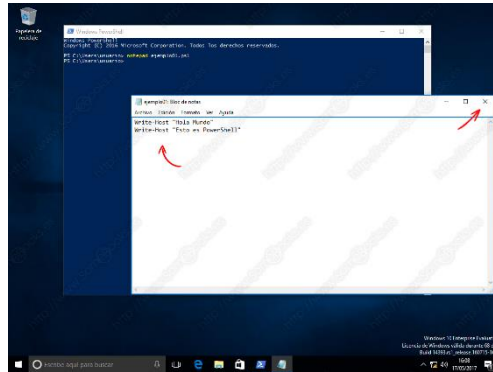
Después de esto, ya estamos listos para comenzar a escribir el contenido de nuestro primer *script*. De momento, nos conformaremos con algo sencillo: Escribir dos líneas de texto en la terminal. En la primera aparecerá el texto *Hola Mundo* (todo un clásico de la programación) y en la segunda *Esto es 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.



Como cabe esperar, el editor nos preguntará si queremos guardar los cambios que acabamos de hacer.

Hacemos clic sobre el botón *Guardar*.

Llamamos *prompt* al texto que utiliza el sistema para indicarnos que está esperando una orden.

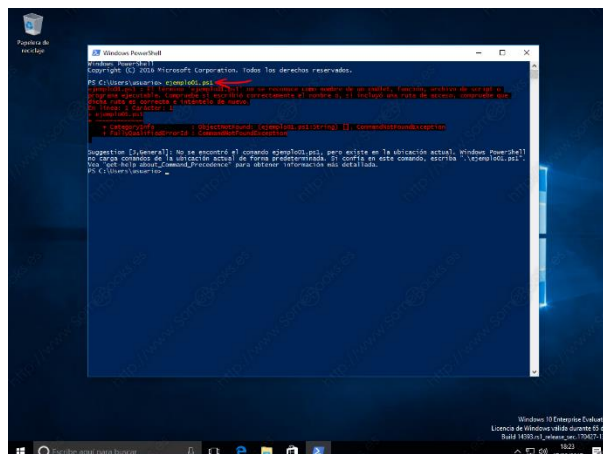
```
PS C:\Users\usuario>
```

Ejecutar el primer 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, si has seguido las indicaciones del apartado anterior, lo más probable es que recibas un error como el de la siguiente imagen.

Sencillamente, no se ha podido encontrar el *script*.



¿Cuál es el motivo? sencillamente, que 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.

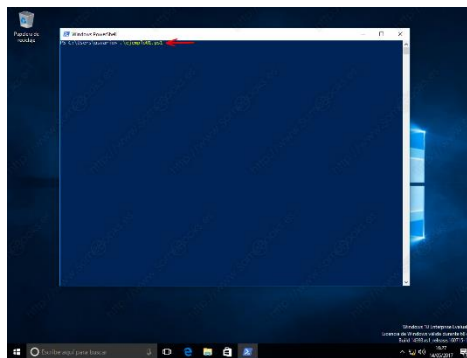
En el ejemplo anterior, hemos guardado el *script* en la carpeta donde nos encontrábamos (*C:\Users\usuario*). Por lo tanto, deberíamos escribir algo parecido a esto:

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

No obstante, si la ruta en cuestión coincide con la *ruta actual* en el momento de ejecutarlo, podemos simplificarlo, escribiendo algo así:

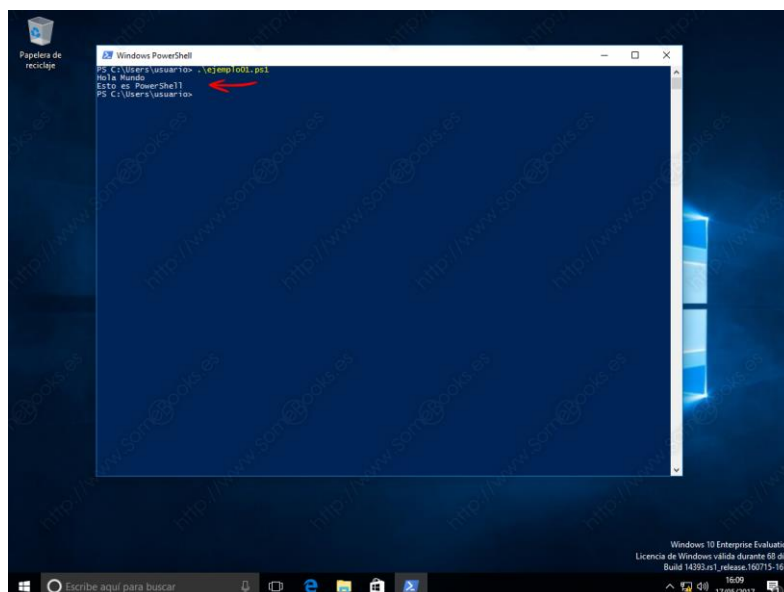
```
.\ejemplo01.ps1
```

Por lo tanto, basta con escribir la orden anterior y pulsar la tecla Intro.



Al hacerlo, se ejecutará el *script*.

Comprobamos que, ahora sí, obtenemos la salida esperada.

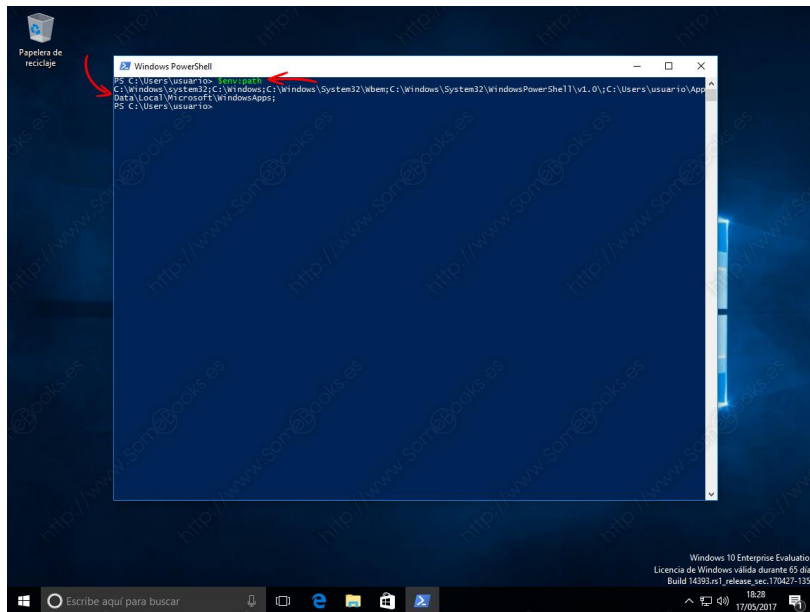


Si necesitas saber cuál es la ruta predeterminada del sistema, basta con

escribir la siguiente orden:

```
$env:path
```

Al pulsar la tecla Intro obtenemos la información que buscamos.



Como ves en la imagen, en mi sistema la salida es esta:

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

Observando más detenidamente verás que, en realidad se trata de varias rutas separadas por caracteres 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;  
;
```

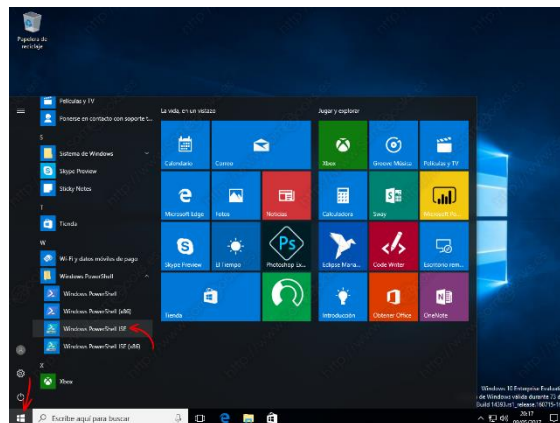
Bastaría con guardar el script en cualquiera de esos directorios para que pudiésemos eliminar la ruta del nombre del archivo.

El entorno de desarrollo integrado Windows PowerShell ISE

Acabamos de ver cómo, con un simple editor de textos básico, podemos crear scripts en PowerShell. Sin embargo, ya vimos en el Punto 1 que PowerShell dispone de un entorno gráfico llamado *PowerShell ISE* (del inglés, *Integrated Scripting Environment*). Su gran ventaja es que integra las tareas relativas a la escritura, depuración y prueba de *scripts*.

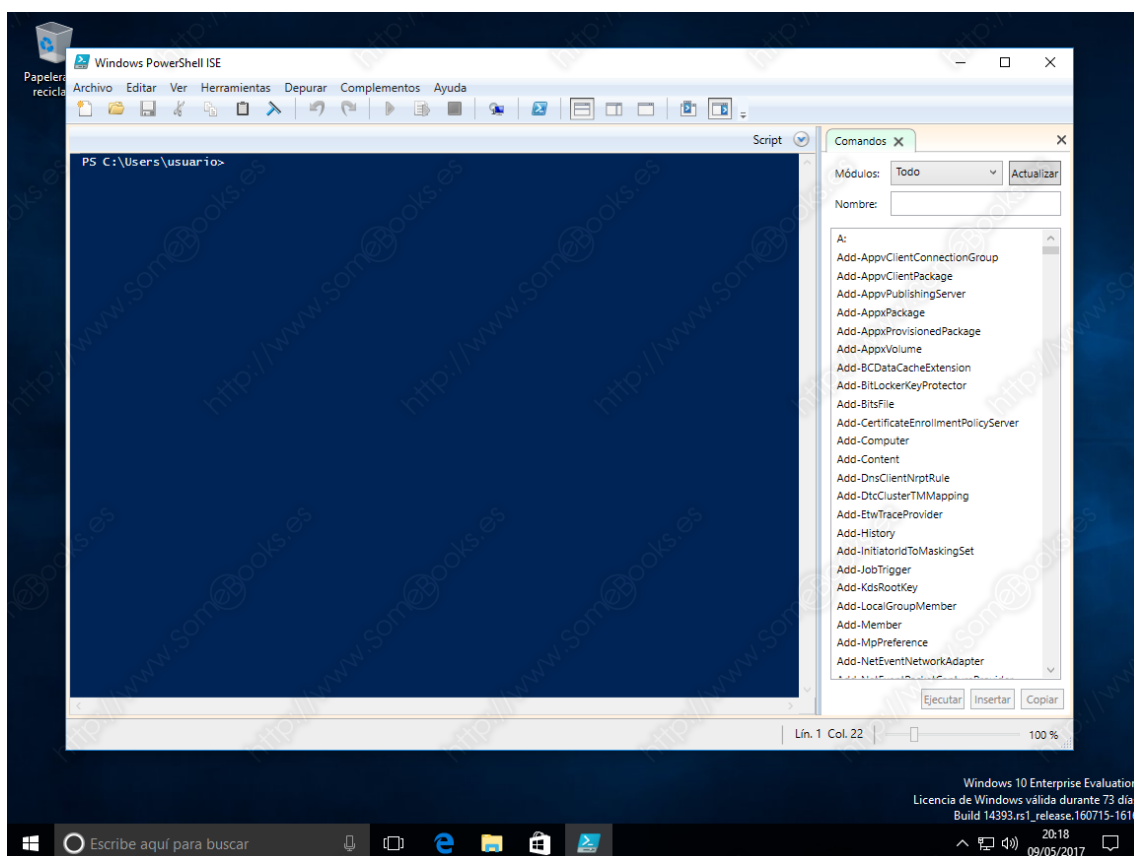
Para ejecutarlo, haremos clic sobre el botón *Inicio* y desplazarnos por el menú hasta la carpeta *Windows PowerShell*.

En su interior, haremos clic sobre el elemento *Windows PowerShell ISE*.



Poco después se abrirá el entorno de desarrollo de *PowerShell*.

A partir de este momento, podremos comenzar a trabajar con las diferentes opciones de la interfaz gráfica.



Comparado con el *Bloc de notas*, *PowerShell ISE* ofrece una gran variedad de opciones que, sobre todo al principio, puede marear un poco. Para evitarlo, vamos a realizar un pequeño recorrido por sus diferentes apartados:

Barra de menú: se encuentra justo debajo del título de la ventana. Contiene todas las opciones disponibles en *PowerShell ISE*.

Barra de herramientas: se encuentra debajo de la barra de menú y contiene las opciones que se utilizan de manera más frecuente.



De izquierda a derecha, encontramos los siguientes elementos

Nuevo, Abrir y Guardar () , que se aplican al *script* en su conjunto.

Cortar, Copiar y Pegar () , que se aplican al texto que tengamos seleccionado.

Borrar panel de consola () , que elimina de la pantalla el resultado de las órdenes ejecutadas con anterioridad.

Deshacer y Rehacer () , que nos permiten actuar sobre la última acción que hayamos realizado.

Ejecutar script, Ejecutar selección y Detener operación () , que facilitan la ejecución de scripts.

Nueva pestaña de PowerShell en remoto () , que permite ejecutar *scripts* en un equipo diferente.

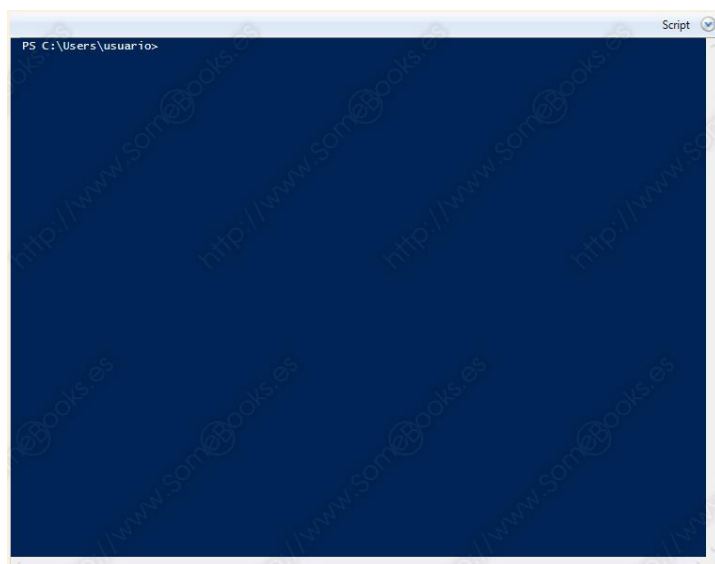
Iniciar PowerShell.exe () , que abre una ventana de PowerShell.

Mostrar panel de scripts arriba, a la derecha o maximizado () , que permiten elegir la ubicación del panel en el que escribiremos los scripts. La opción predeterminada es la primera.

Mostrar ventana de comando y Mostrar complemento de

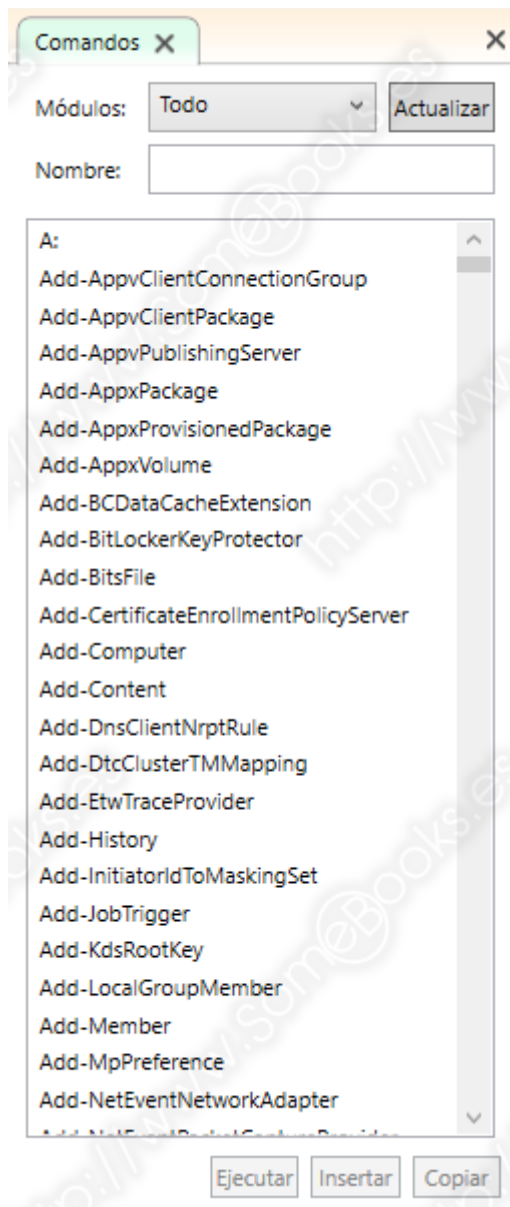
comando () , el primero muestra una ventana flotante con los cmdlets disponibles y el segundo lo hace en forma de panel en el lateral derecho (ver más abajo el punto *Comandos*)

Consola: Es donde obtendremos la salida de los *scripts* que ejecutemos. También podemos usarla para ejecutar comandos directamente, como en cualquier ventana de *PowerShell*.

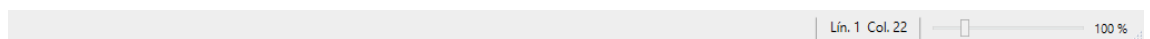


Comandos: Contiene la lista completa de *cmdlets* disponibles. Cuando elegimos uno, aparece una descripción. También podemos definir la sintaxis completa de una orden para ejecutarla en la *Consola* cuando lo consideremos necesario.

Usando los botones de la parte inferior (), podemos ejecutar el comando seleccionado con los valores que hayamos indicado previamente, insertarlo en la Consola para poder editarlo o copiarlo en el portapapeles para usarlo más adelante.



Barra de estado: Cuando estamos escribiendo un *script*, ofrece información sobre la ubicación actual del cursor (número de línea y columna) para facilitar la vuelta a ese punto en el futuro. También tenemos un indicador de zoom, que nos permite ampliar o reducir la vista de la ventana.

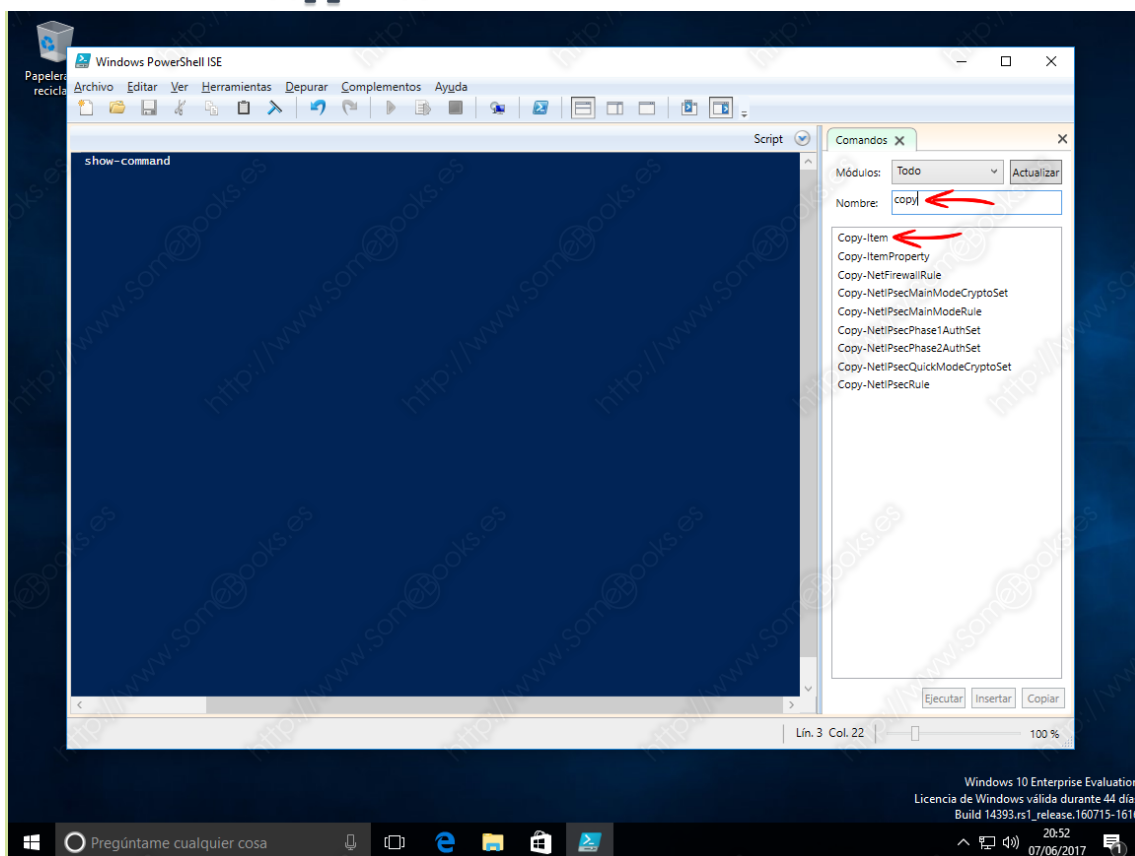


Crear comandos de forma interactiva en PowerShell ISE

Imagina que, al final del día, necesitas copiar tu trabajo en una *memoria USB*, para asegurarte de ponerlo a salvo de sorpresas desagradables. Los archivos que te interesan tienen extensiones .odt y .ods. por lo que, a diario, escribirías una orden como esta:

```
Copy-Item -Path c:\Users\Usuario\trabajo\* -Include "*.odt", "*.ods" -Destination e:\trabajo
```

En estos casos, la solución puede pasar por utilizar el panel *Comandos* para que nos asista en la sintaxis de **Copy-Item**. Para probar cómo funciona, puedes comenzar escribiendo el nombre del *cmdlet* en el cuadro *Nombre*. Verás que en la lista inferior pasan a mostrarse únicamente los *cmdlets* que contienen el texto que has escrito. Cuando localices **Copy-Item** haz clic sobre él.

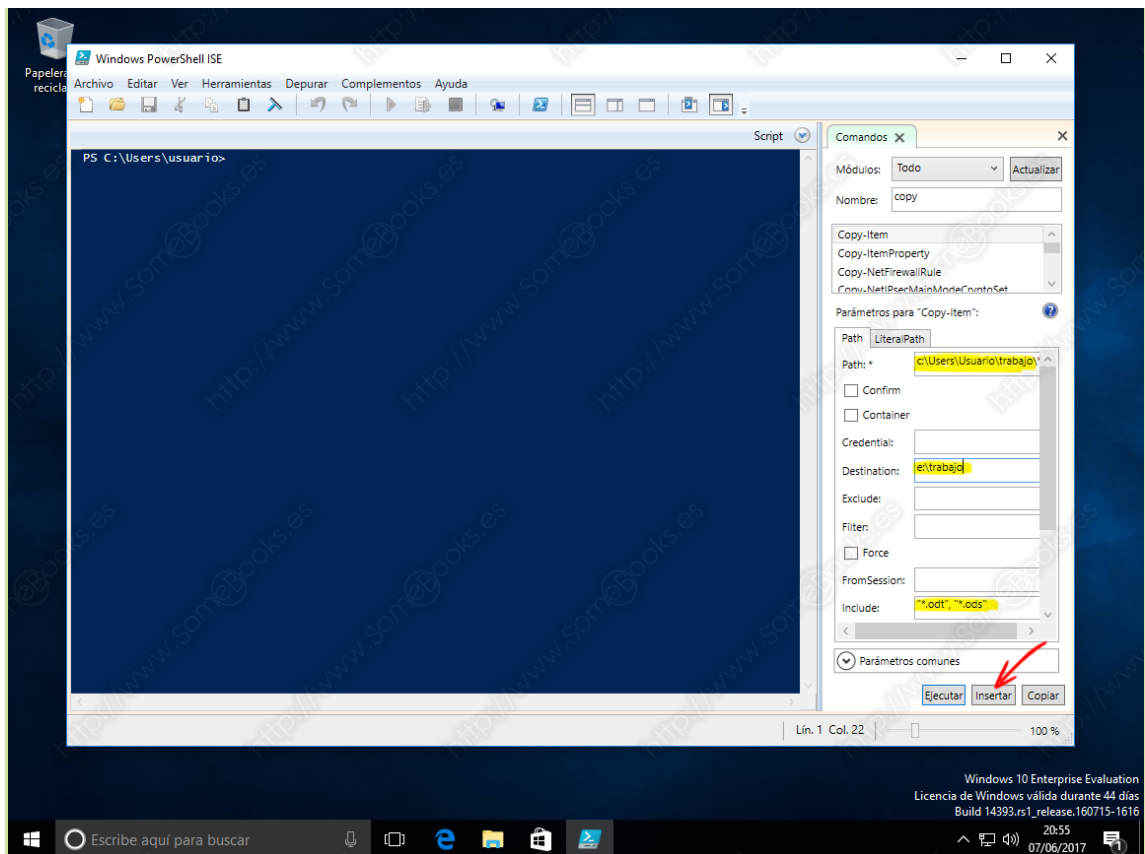


Al hacerlo, se abrirá un nuevo cuadro en la parte inferior del panel ofreciéndonos el botón *Mostrar detalles* para importar el módulo donde se encuentra el *cmdlet*.

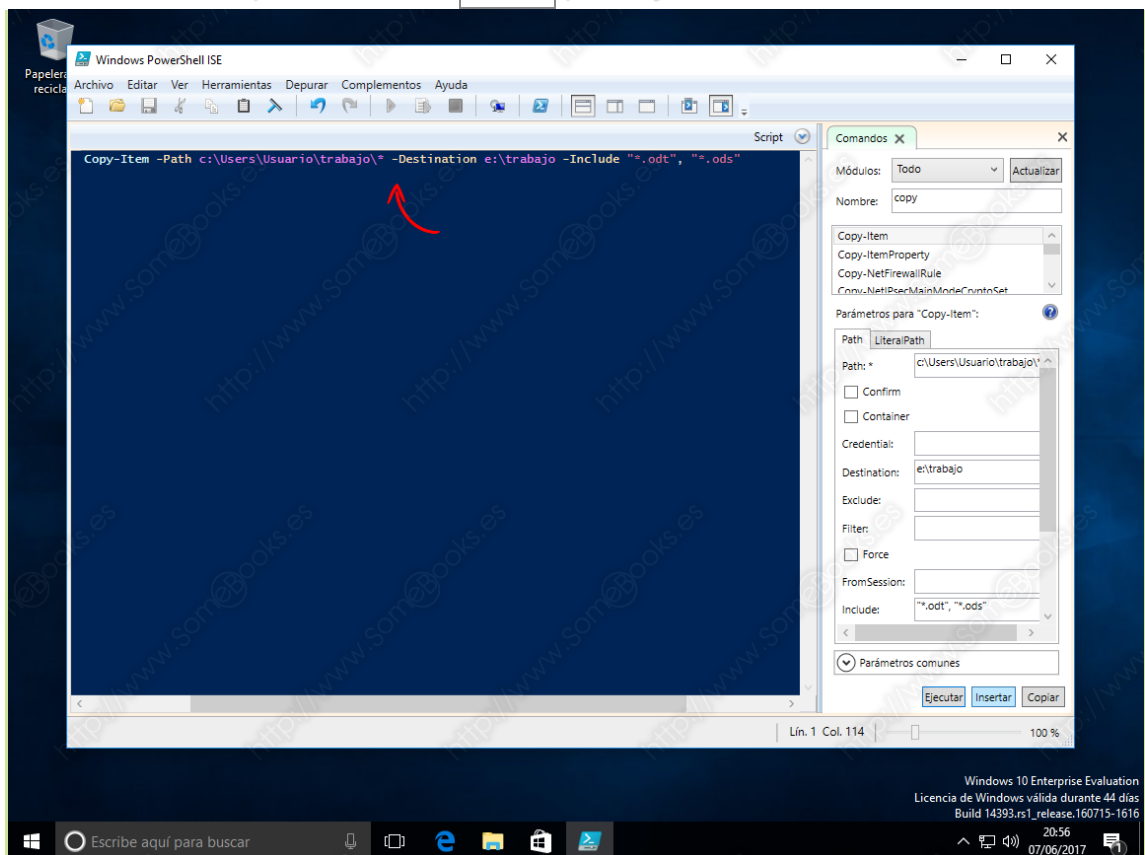
Hacemos clic en el botón *Mostrar detalles*.

El resultado será un pequeño formulario donde podemos escribir el contenido para los argumentos de **Copy-Item**. Cuando hayamos completado los valores adecuados, hacemos clic sobre el botón *Insertar*.

Cuando hayamos terminado, hacemos clic sobre el botón *Insertar*.

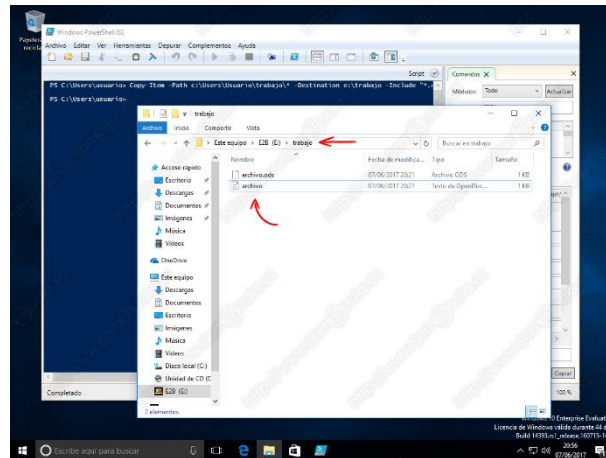


Al hacerlo, aparecerá el **cmdlet** completo en la **Consola**, sin errores sintácticos y sin que hayamos necesitado recordar la sintaxis completa. Ya sólo nos falta pulsar la tecla **Intro** para ejecutarlo.



Si ahora abrimos el dispositivo de destino, podemos comprobar que la copia se ha realizado con éxito.

Los archivos se han copiado en su destino.



Crear un nuevo script en PowerShell ISE

Para terminar este Punto, vamos a aprender a crear un nuevo *script* en *PowerShell*, ejecutarlo, guardarlo y más adelante recuperarlo para volver a utilizarlo.

Para conseguirlo, usaremos el mismo *script* que creamos al principio del Punto:

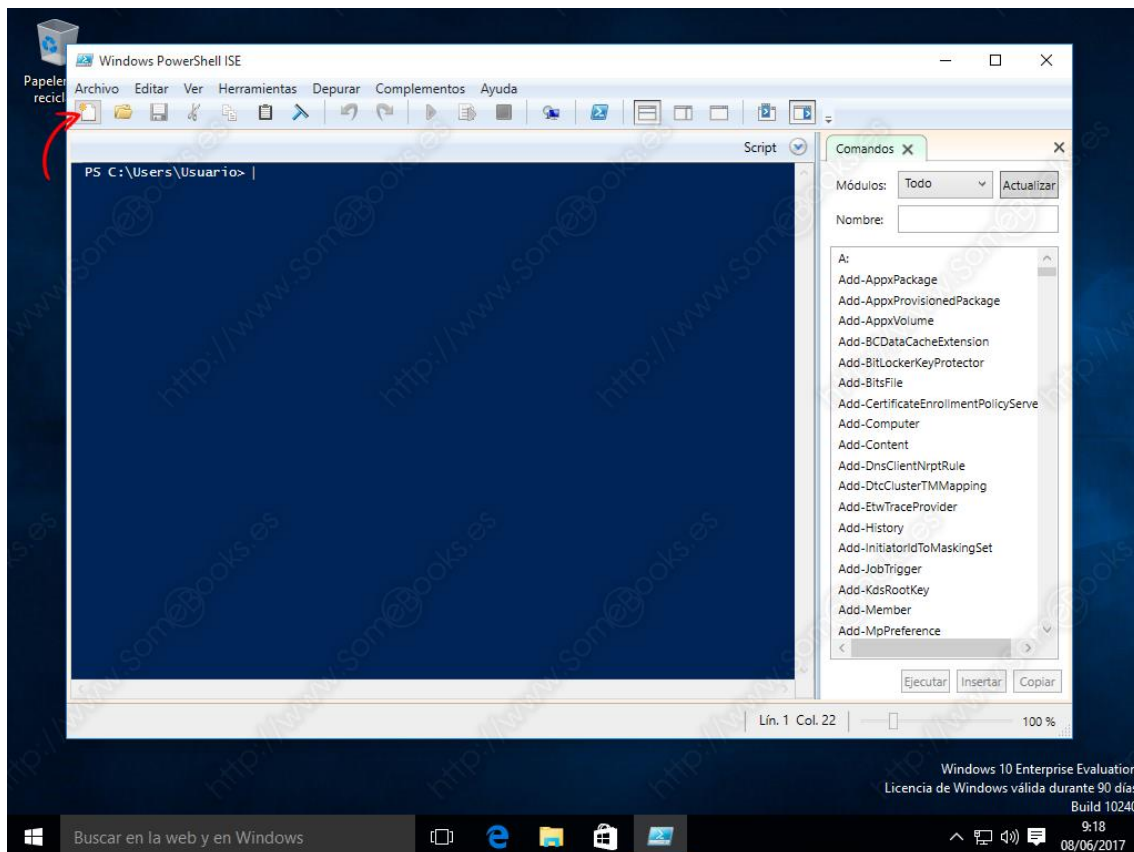
```
Write-Host "Hola Mundo"
```

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

Vamos paso a paso...

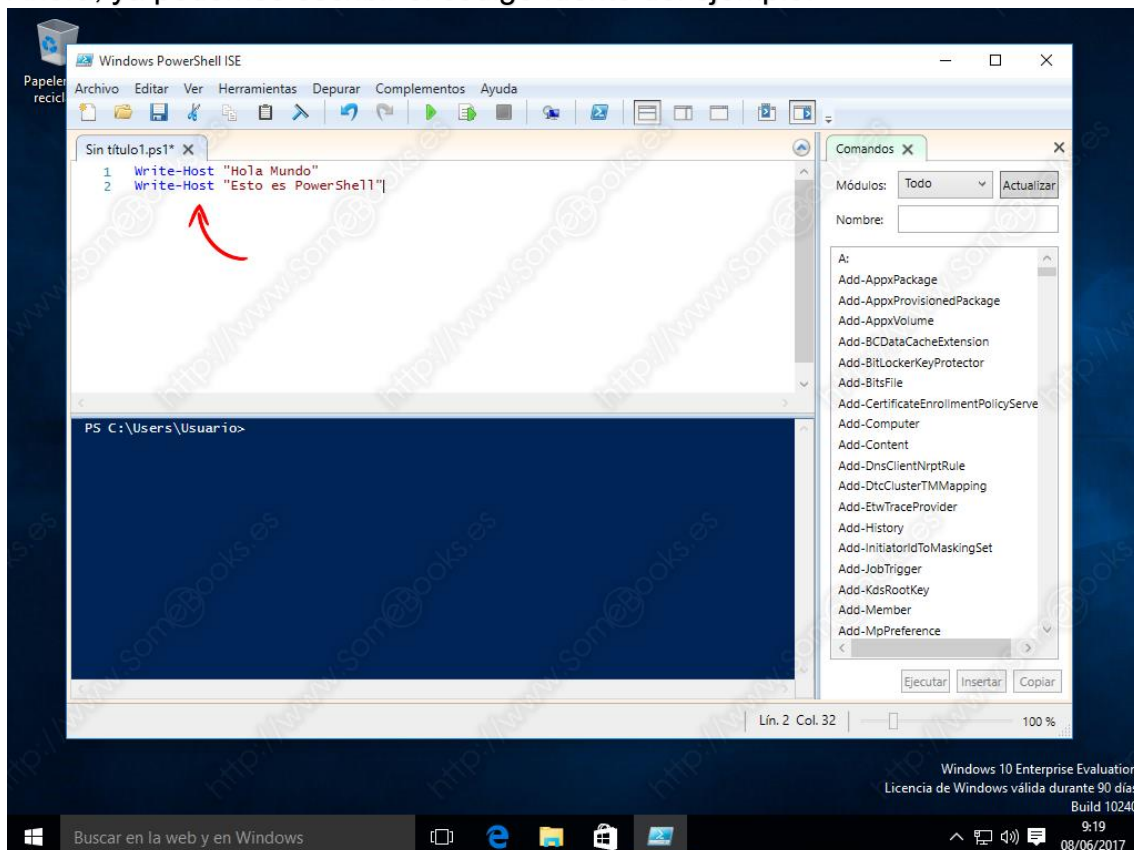
Lo primero será hacer clic sobre el botón *Nuevo* de la barra de herramientas



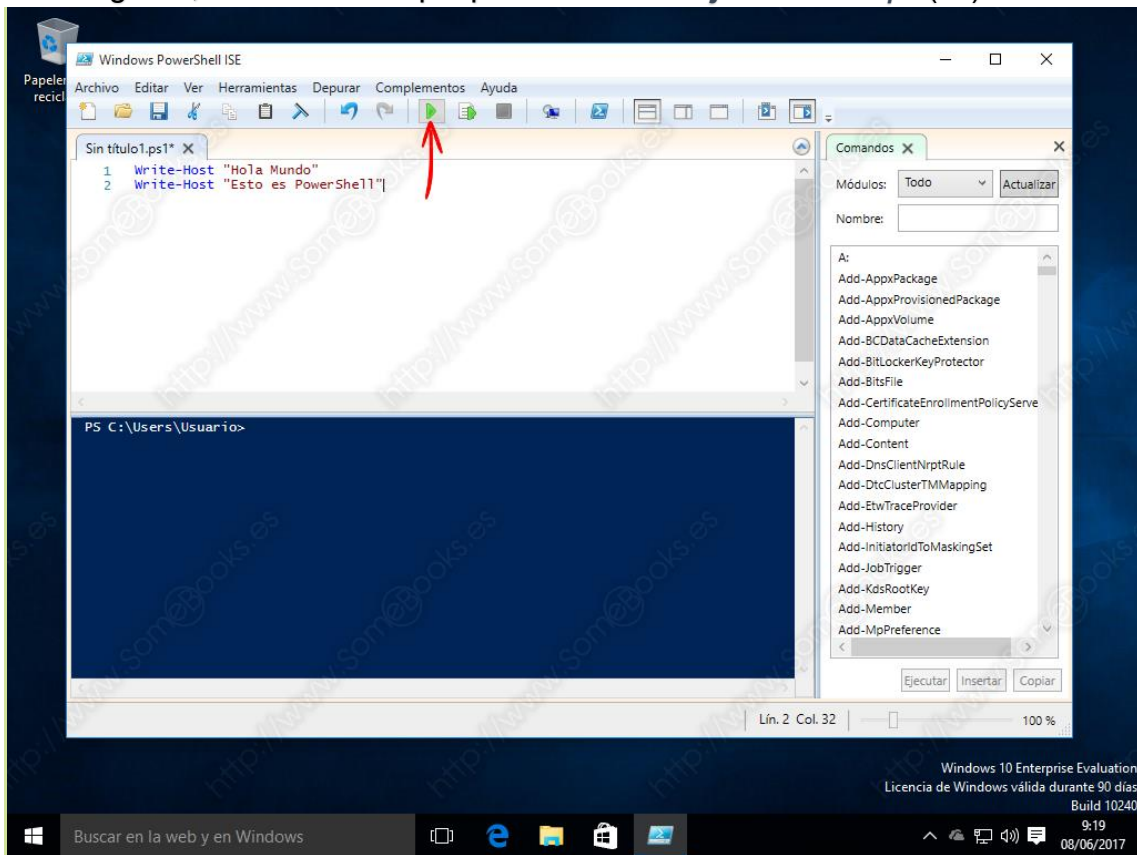


El resultado es que, la mitad superior del área de la *Consola*, es ocupada por el *Editor*.

Ahora, ya podemos escribir el código fuente del ejemplo.

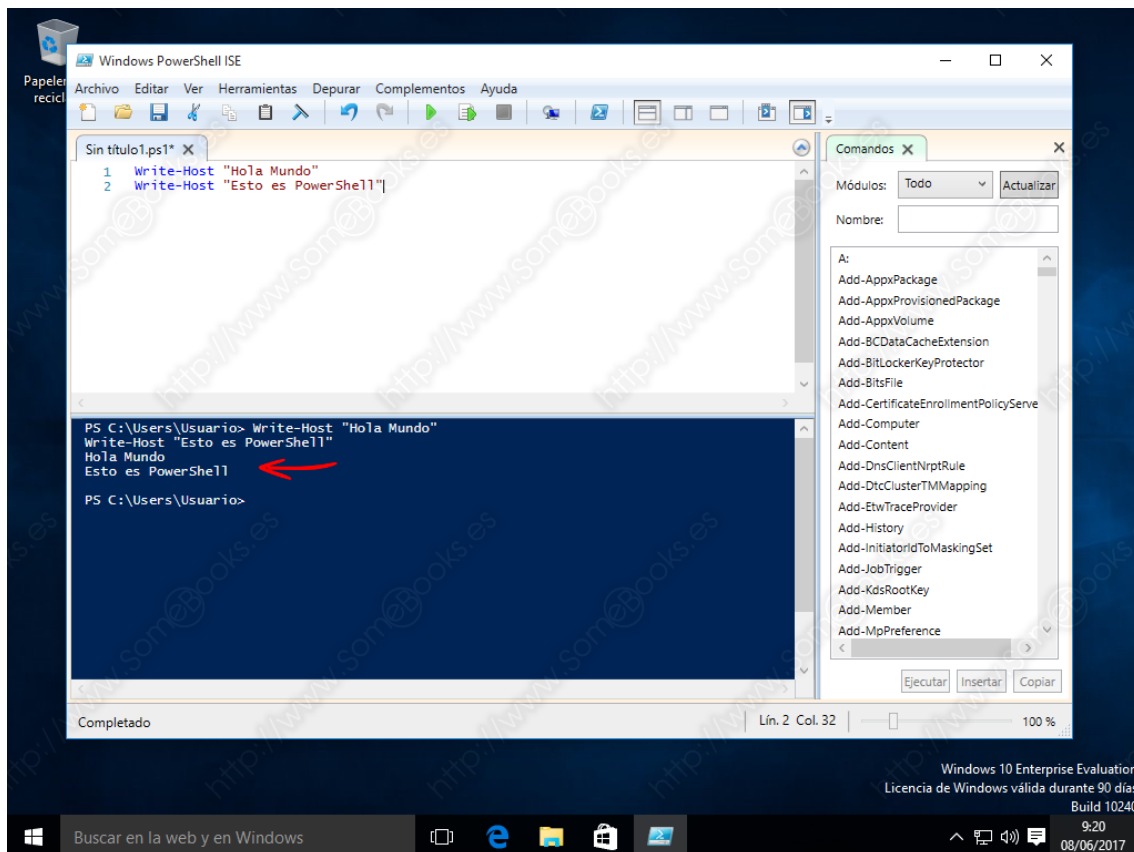


Cuando terminemos de escribirlo, podemos comprobar su funcionamiento. Para lograrlo, sólo tenemos que pulsar el botón *Ejecutar script* (▶).



Al hacerlo, se ejecuta el *script* y el resultado aparece en el área de *Consola*.

Comprobamos que el script se ejecuta correctamente.



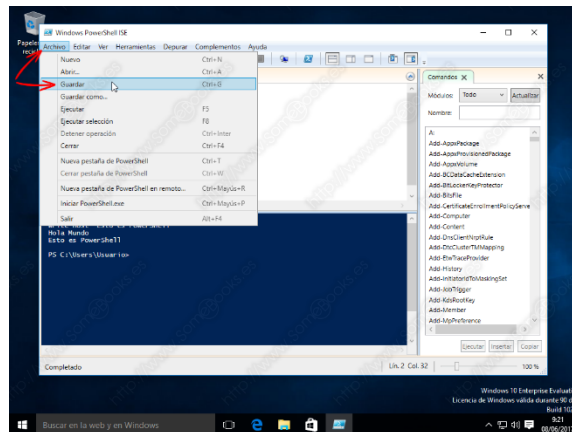
Guardar un script en PowerShell ISE para usarlo en el futuro.

Es normal que, una vez que hayamos comprobado el buen funcionamiento de un script, queramos conservarlo para usarlo en el futuro. De esta forma, podemos ahorrarnos volver a escribirlo.

Para conseguirlo, podemos recurrir al botón *Guardar* (📁) de la *barra de herramientas*, o elegir *Archivo > Guardar* en la *barra de menú*. Incluso tenemos la opción *Guardar como...*, para guardar una copia modificada de un *script* que ya teníamos guardado previamente.

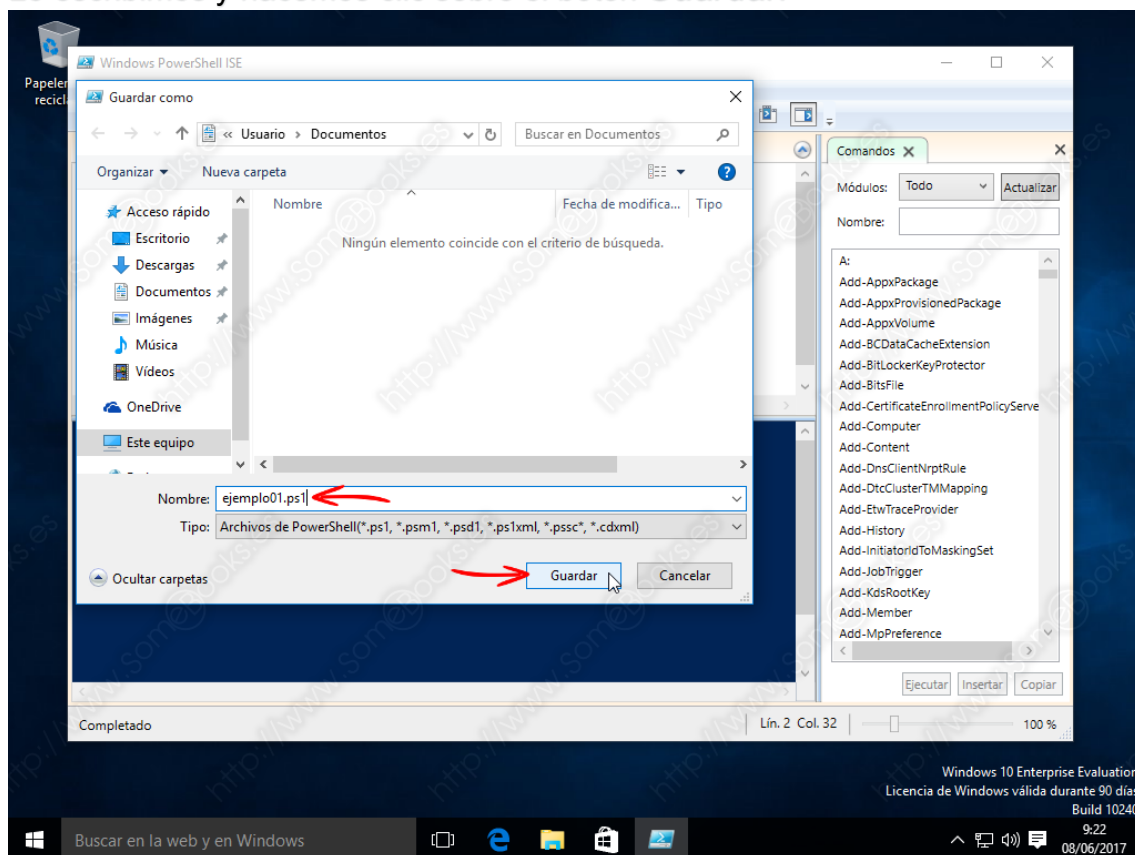
En el caso de que el *script* no exista aún, las tres opciones anteriores funcionarán de la misma forma. Pero cuando ya exista el archivo, las dos primeras sobrescribirán el contenido original con las modificaciones que hayamos hecho.

Por ejemplo, elegimos *Archivo > Guardar*.



Esto ocasionará que aparezca una nueva ventana donde podemos escribir el nombre que queremos darle al *script*.

Lo escribimos y hacemos clic sobre el botón *Guardar*.



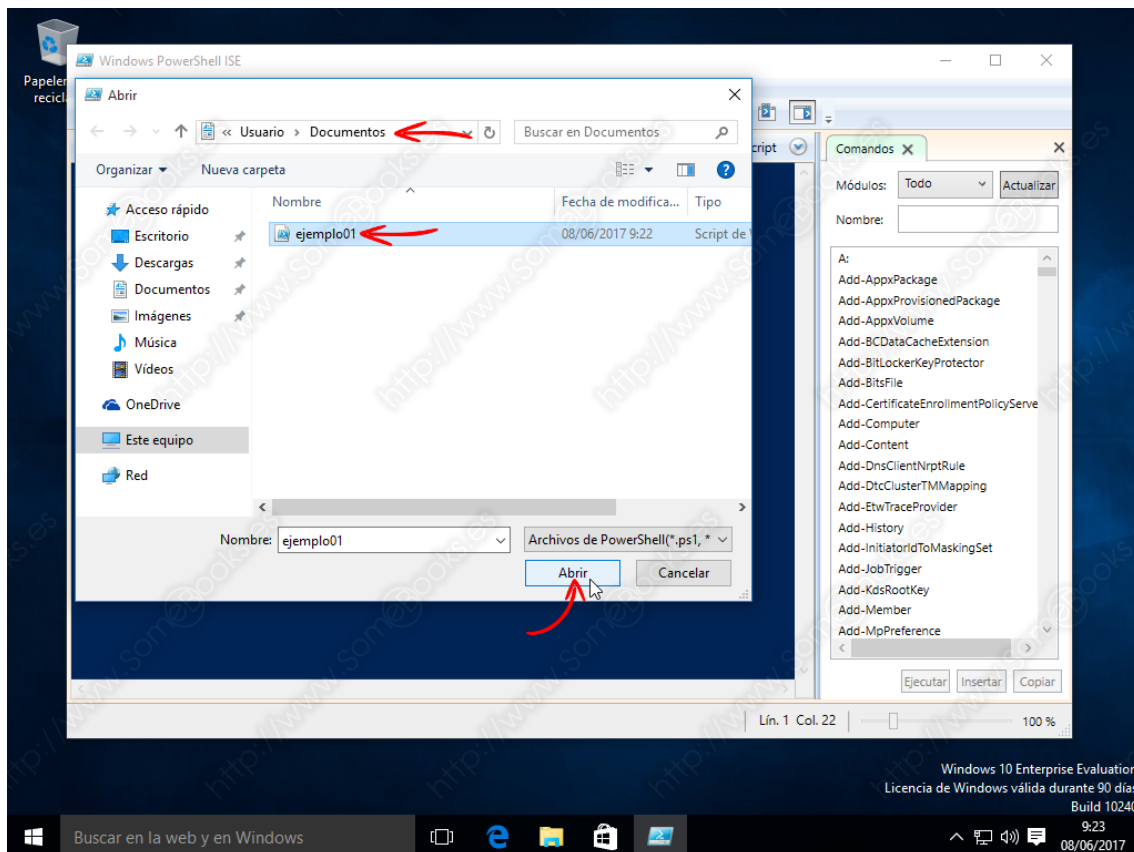
Abrir en PowerShell ISE un script que ya existe

Quando necesitamos recuperar un archivo que ya tengamos guardado en el disco, bastará con elegir la opción *Archivo > Abrir...* en la *barra de menú*.

También podemos usar el botón *Abrir* (📁) de la *barra de herramientas*.

De forma parecida a lo que ocurrió en el apartado anterior, ahora aparecerá una nueva ventana. Ésta nos permitirá navegar por el contenido del disco hasta localizar el archivo que nos interesa.

Quando lo encontremos, basta con hacer doble clic sobre él, o seleccionarlo y hacer clic sobre el botón *Abrir*.



Un momento después, el contenido del script estará ocupando la ventana del editor de PowerShell ISE.

... Y podemos trabajar con él como si acabáramos de escribirlo.

Punto 3: 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*. Como veremos después, el valor almacenado puede ser de naturaleza diferente (numérico, textual, lógico, ...)

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.

Usar variables en PowerShell

Para definir una variable en **PowerShell** sólo tenemos que nombrarla, aunque 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:

```
$ho1a = "¡Hola, Mundo!"
```

Un par de observaciones sobre el ejemplo anterior:

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 nombres válidos para variables podrían ser estos:

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

Podríamos decir que la forma de definir variables que hemos explicado 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
```

Observa que no hemos utilizado el símbolo de dólar (\$) al principio, pero sí lo incluiremos al utilizarla.

Si queremos una sintaxis más descriptiva, podemos incluir argumentos:

```
New-Variable -Name Resultado_1
```

Incluso podríamos asignar un valor inicial:

```
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:

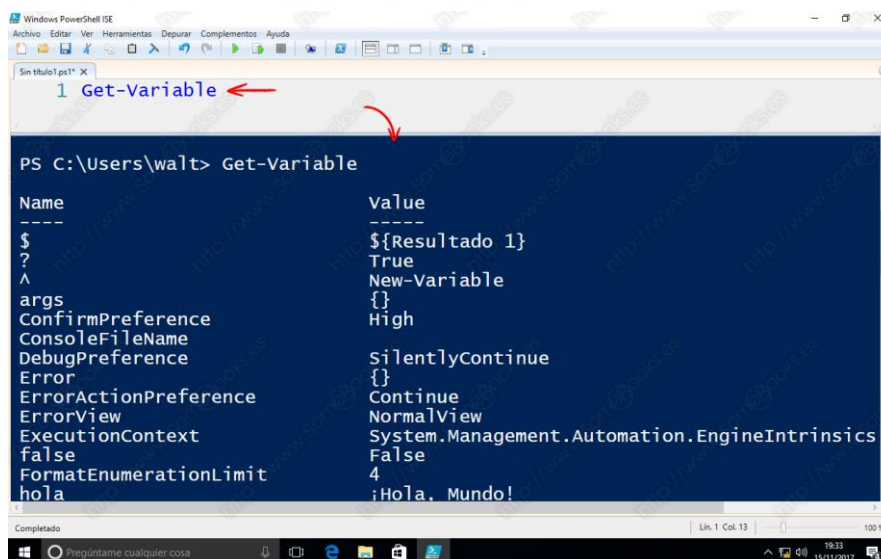
```
New-Variable -Name Impuesto -Value 21 -Option ReadOnly
```

Formalmente, las variables que no cambian a lo largo del *script*, reciben el nombre de *constantes*.

Para profundizar sobre el uso de **New-Variable** puedes recurrir a la página <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/new-variable>.

En cualquier momento puedes obtener una lista completa de las variables que se hayan definido hasta ese momento. Para lograrlo, basta con utilizar el *cmdlet* **Get-Variable**.

Salida del *cmdlet* **Get-Variable**.



Como ves en la imagen anterior, se muestran tanto las definidas por el usuario como las predefinidas por *PowerShell*

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.

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. Veamos un ejemplo:

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

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

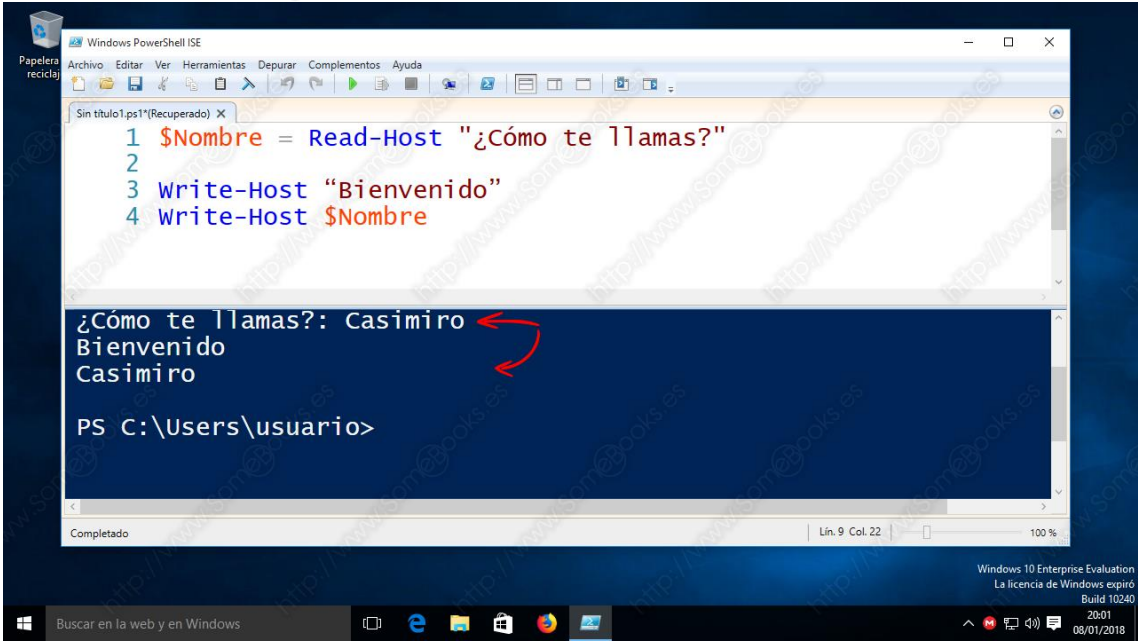
```
PS C:\Users\usuario>  
$Nombre = Read-Host "¿Cómo te llamas?"  
¿Cómo te llamas?: |
```

Mostrar información al usuario

Una vez que hemos visto cómo pedir información al usuario, el siguiente paso es aprender a ofrecérsela. 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
```

El resultado sería algo así:



```
Sin título1.ps1*(Recuperado) X  
1 $Nombre = Read-Host "¿Cómo te llamas?"  
2  
3 Write-Host "Bienvenido"  
4 Write-Host $Nombre  
  
¿Cómo te llamas?: Casimiro  
Bienvenido  
Casimiro  
  
PS C:\Users\usuario>  
  
Completado | Lin. 9 Col. 22 | 100 %
```

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

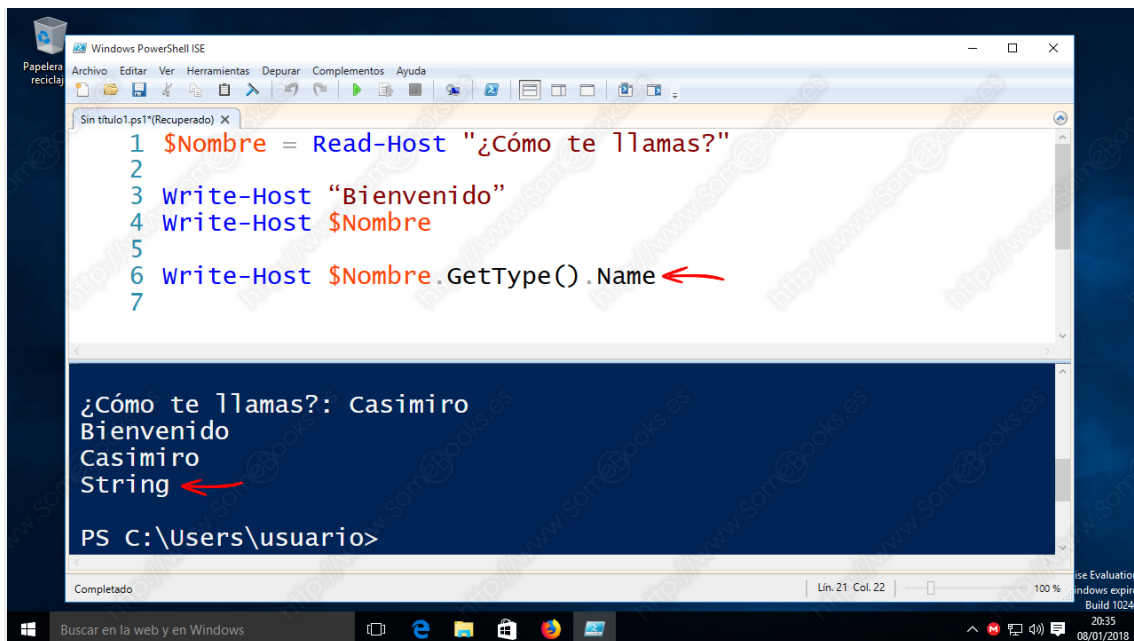
En los ejemplos anteriores hemos visto únicamente cómo podemos guardar texto en una variable. Sin embargo son muchos más los tipos de datos que pueden contener. En la tabla siguiente te mostramos los más habituales:

Tipos de datos		
Tipo de dato	Descripción	Rango de valores
[string]	Cadena de caracteres Unicode	Depende del contenido
[char]	Un solo carácter Unicode de 16 bits	-32768 a 32767
[byte]	Un solo 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 (simple precisión)	Rango negativo: -3.402823E38 a -1.401298E-45 Rango positivo: 1.401298E-45 a 3.402823E38
[double]	Número en coma flotante de 64 bits (doble precisión)	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 ó False
[array]	Objeto que representa un array de valores	Depende del contenido
[hashtable]	Objeto que representa una tabla Hash	Depende del contenido
[xml]	Objeto con contenido xml	Depende del contenido

Más adelante iremos aprendiendo cómo manejar la mayoría de ellos. Si necesitas saber el tipo de dato al que pertenece una determinada variable, puedes utilizar una expresión como esta:

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

... Y si lo aplicamos al ejemplo anterior, obtendríamos la siguiente salida:



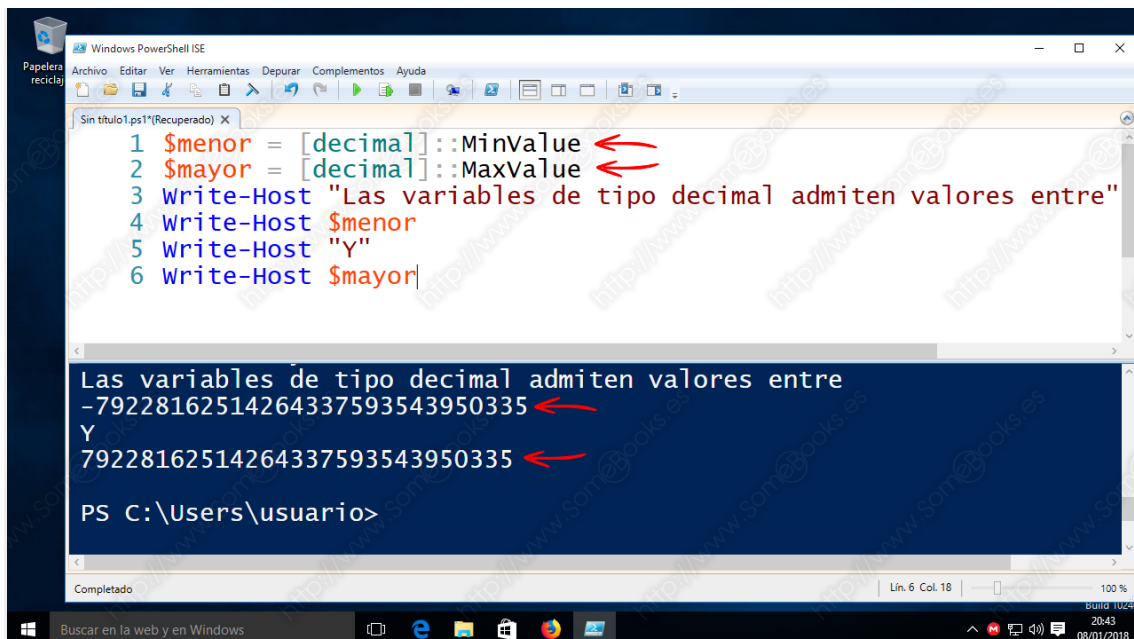
```
Sin título1.ps1 (Recuperado) X
1 $Nombre = Read-Host "¿Cómo te llamas?"
2
3 Write-Host "Bienvenido"
4 Write-Host $Nombre
5
6 Write-Host $Nombre.GetType().Name
7

¿Cómo te llamas?: Casimiro
Bienvenido
Casimiro
String
PS C:\Users\usuario>
```

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

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

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



```
Sin título1.ps1 (Recuperado) X
1 $menor = [decimal]::MinValue
2 $mayor = [decimal]::MaxValue
3 Write-Host "Las variables de tipo decimal admiten valores entre"
4 Write-Host $menor
5 Write-Host "Y"
6 Write-Host $mayor

Las variables de tipo decimal admiten valores entre
-79228162514264337593543950335
Y
79228162514264337593543950335
PS C:\Users\usuario>
```

Como cabe esperar, 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 tratara de amoldarlo a la variable.

Definición implícita:

Como hemos dicho más arriba, el tipo de dato de estas variable se establece en el momento de asignarle valor. De hecho, ya vimos al principio una definición de esta clase:

```
$hola = "¡Hola, Mundo!"
```

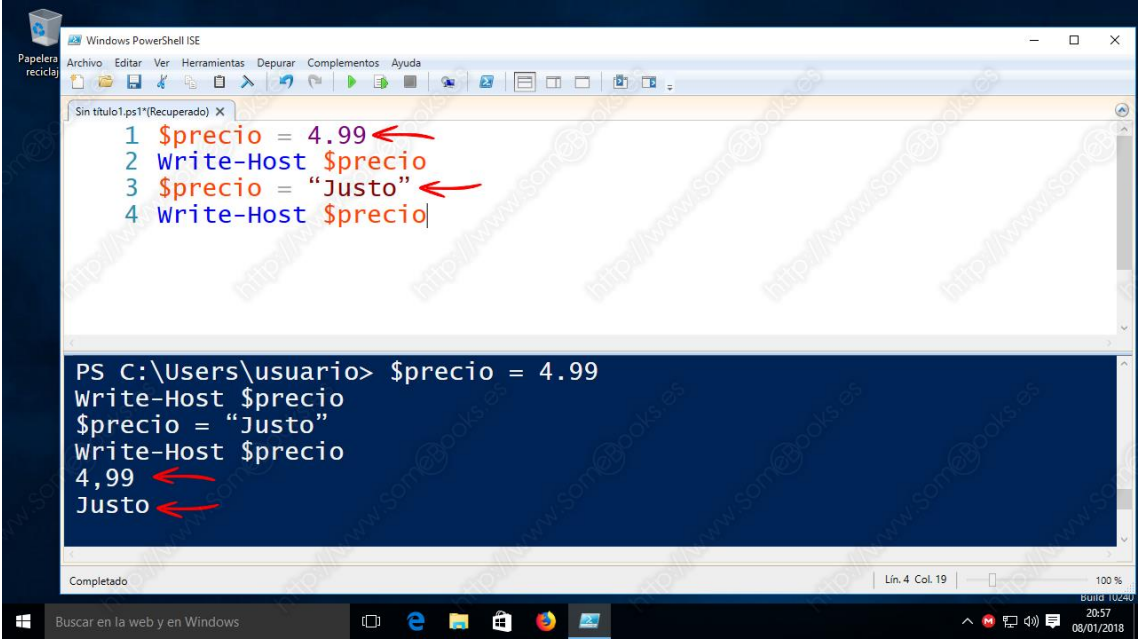
En este caso, la variable `$hola` se crea en el momento de usarla y se le asigna un valor de tipo texto (al que llamaremos *string* o *cadena de caracteres*). Este método también se utiliza para crear datos de otros tipos. Por ejemplo, en el siguiente ejemplo creamos una variable numérica (en particular, 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 si experimentar ningún tipo de error:

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

... Y si lo comprobamos, obtendremos una salida como esta:



The screenshot shows the Windows PowerShell ISE interface. The script editor contains four lines of code: `1 $precio = 4.99`, `2 Write-Host $precio`, `3 $precio = "Justo"`, and `4 Write-Host $precio`. Red arrows point from the first and third lines to the output console. The output console shows the results: `PS C:\Users\usuario> $precio = 4.99`, `Write-Host $precio` outputs `4,99`, `$precio = "Justo"`, and `Write-Host $precio` outputs `Justo`. The status bar at the bottom indicates 'Completado' and 'Lin. 4 Col. 19'.

Definición explícita:

En este caso, 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.

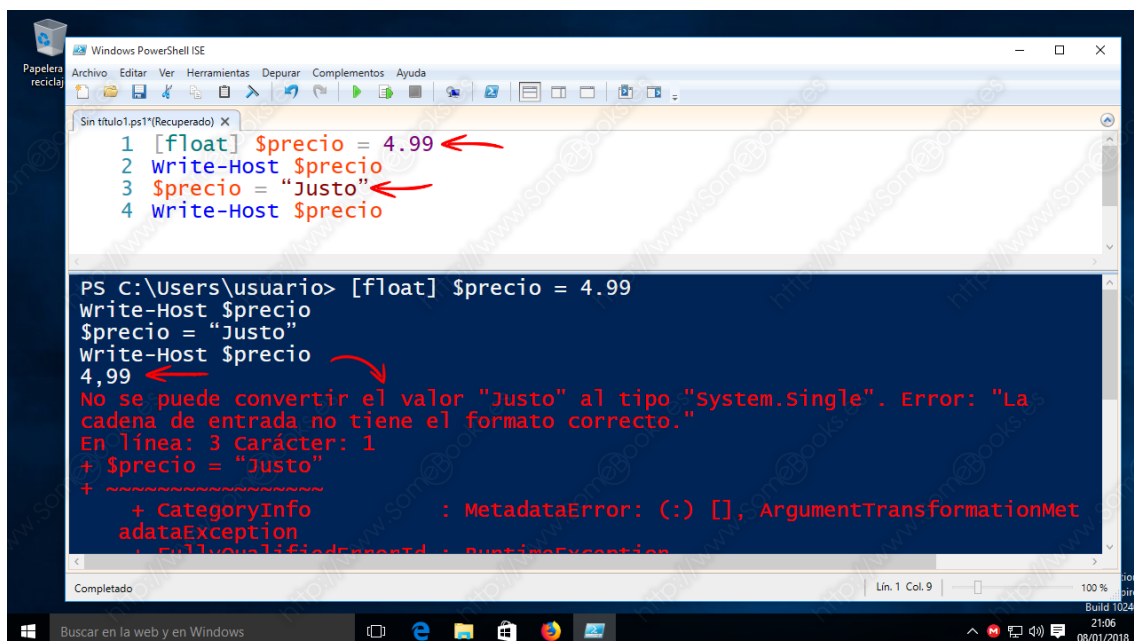
Por ejemplo, como hemos dicho antes, la variable `$precio` del ejemplo anterior será de tipo *double*, por lo que ocupará en memoria 64 bits. Si nos parece excesivo este gasto de memoria por la naturaleza de los datos que estamos almacenando, al definir la variable, podemos especificar el tipo de dato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

```
[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. Observa la imagen siguiente si tratamos de repetir ahora el ejemplo anterior:



```
Sin título1.ps1*(Recuperado) X
1 [float] $precio = 4.99
2 Write-Host $precio
3 $precio = "Justo"
4 Write-Host $precio

PS C:\Users\usuario> [float] $precio = 4.99
write-host $precio
$precio = "Justo"
write-host $precio
4.99
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: (:) [], ArgumentTransformationMet
+ FullyQualifiedErrorId : RuntimeException
```

Especificar el tipo de un dato

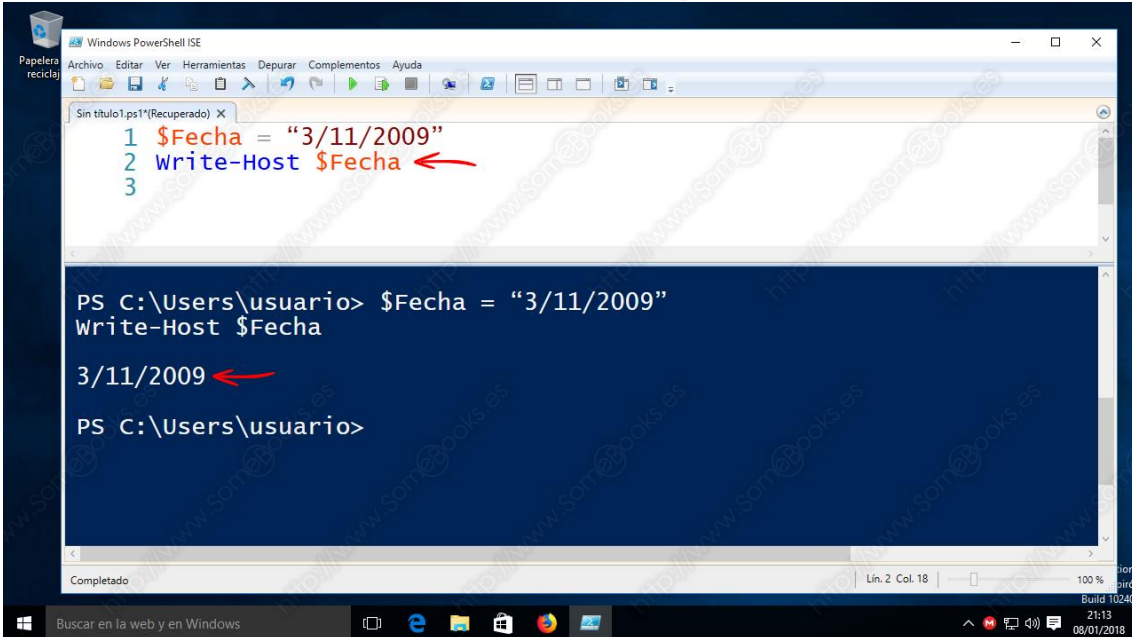
Otra alternativa que tenemos a nuestra disposición consiste en indicar el tipo al que pertenece el propio dato. De este modo, podríamos lograr las ventajas de las dos opciones anteriores: tener una variable que pueda cambiar su tipo y usar el espacio de memoria adecuado para el dato que estamos guardando. 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. Para entenderlo, observemos el siguiente ejemplo:

```
$Fecha = "3/11/2009"
```

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:



The screenshot shows a Windows PowerShell ISE window with a menu bar (Archivo, Editar, Ver, Herramientas, Depurar, Complementos, Ayuda) and a toolbar. The script editor contains three lines: 1. `$Fecha = "3/11/2009"`, 2. `Write-Host $Fecha` (with a red arrow pointing to it), and 3. (empty). The console window below shows the execution: `PS C:\Users\usuario> $Fecha = "3/11/2009"`, `Write-Host $Fecha`, and the output `3/11/2009` (with a red arrow pointing to it). The status bar at the bottom indicates "Completado", "Lín. 2 Col. 18", and "100 %". The taskbar at the bottom shows the Windows logo, search bar, and various application icons.

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]"3/11/2009"
```

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

```
Sin título1.ps1*(Recuperado) X
1 $Fecha = [datetime]"3/11/2009"
2 Write-Host $Fecha
3

PS C:\Users\usuario> $Fecha = [datetime]"3/11/2009"
Write-Host $Fecha

11/03/2009 0:00:00

PS C:\Users\usuario>
```

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, cuando escribimos lo siguiente:

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

Obtendríamos un resultado como este:

```
Sin título1.ps1*(Recuperado) X
1 $precio = 4.99
2 Write-Host ([int]$precio)
3

PS C:\Users\usuario> $precio = 4.99
Write-Host ([int]$precio)

5

PS C:\Users\usuario>
```

Como puedes comprobar, *PowerShell* nos ofrece el entero más próximo al valor dado.
Cuando esto no sea posible, sencillamente se producirá un error de ejecución.


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

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

```
1 $precio = "Justo"
2 Write-Host ([int]$precio)
3
```

A red arrow points to the `[int]` type cast in line 2. Below the script editor, the PowerShell console displays the execution of the script, resulting in a runtime error:

```
PS C:\Users\usuario> $precio = "Justo"
Write-Host ([int]$precio)

No se puede convertir el valor "Justo" al tipo "System.Single". Error:
cadena de entrada no tiene el formato correcto.
En línea: 1 Carácter: 1
+ $precio = "Justo"
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationException
+ FullyQualifiedErrorId : MetadataError: (:) [], ArgumentTransformationException
```

A red arrow points to the error message. The status bar at the bottom indicates "Completado" (Completed) and "Lin: 3 Col: 1".

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

básicos: *char, byte, int, long, single, double, decimal, datetime* y *bool*.

Punto 4: Operaciones básicas con variables

En el Punto anterior hemos hablado del concepto de variable y de cómo realizar un uso básico de ellas para guardar información. También aprendimos que la información que almacenamos puede ser de tipos diferentes y que las variables pueden amoldarse a tipos de datos cambiantes u obligar a que su contenido pertenezca a un tipo de dato en particular.

Página 45 | 101

a realizar operaciones con el contenido de las variables y, si es necesario, que el destino de ese resultado sea también una variable.

Para lograrlo, disponemos de un conjunto de operadores que nos permitirán realizar dichos cálculos. Además, según el tipo de dato sobre el que pretendamos actuar, deberemos utilizar unos u otros.

Operar con variables de tipo texto

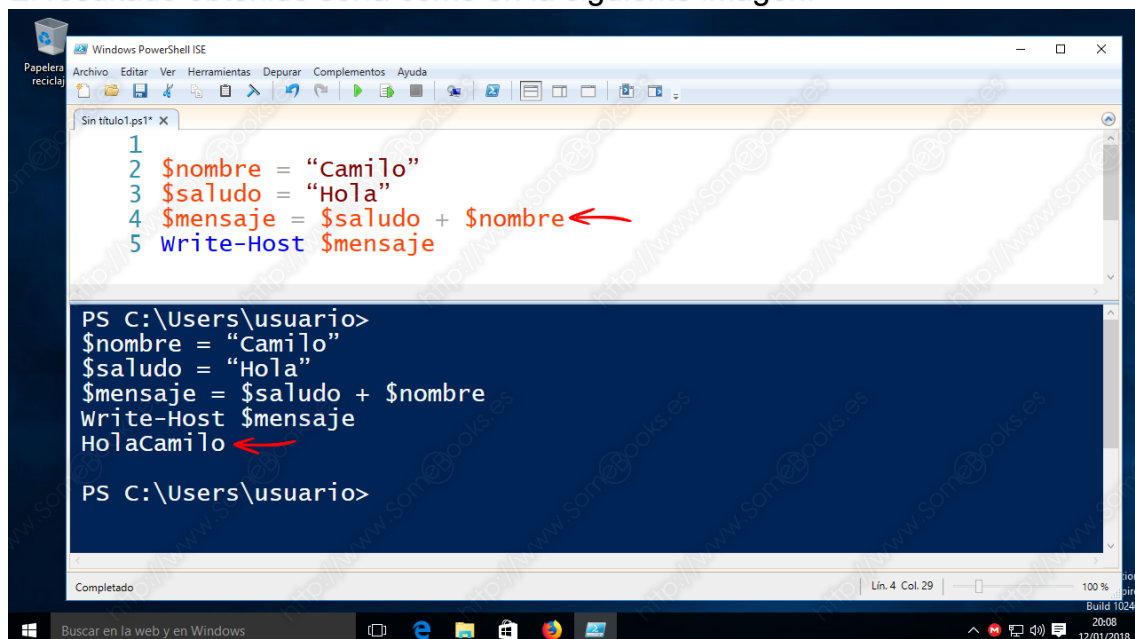
Ya conocemos el operador más sencillo que podemos usar con variables de texto: El operador de asignación. Con él, asignamos el contenido de la expresión que haya a su derecha a la variable indicada a la izquierda. Además, podemos utilizarlo con variables de cualquier tipo:

```
$nombre = "Camilo"
```

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

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

El resultado obtenido sería como en la siguiente imagen:



Observa que 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.

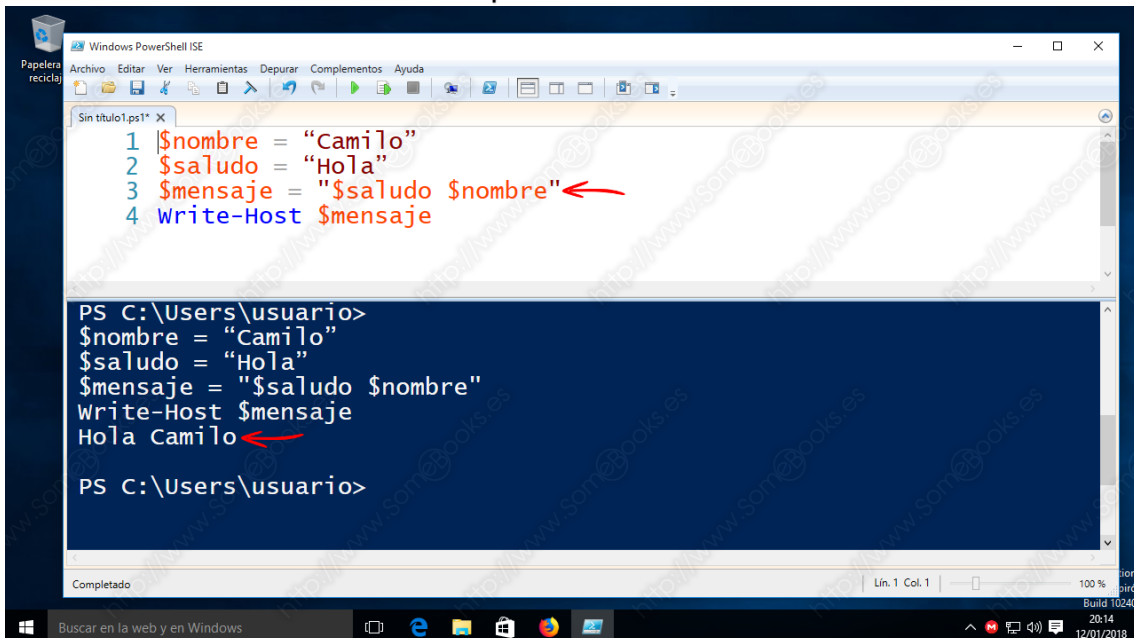
Únicamente se limita a *pegar* el contenido de ambas variables en una sola. Para solucionarlo, podríamos crear una expresión como esta:

```
$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:

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

... Y el resultado obtenido sería equivalente:

A screenshot of the Windows PowerShell ISE interface. The script editor at the top shows four lines of code: 1. \$nombre = "Camilo", 2. \$saludo = "Hola", 3. \$mensaje = "\$saludo \$nombre" (with a red arrow pointing to the double quotes), and 4. Write-Host \$mensaje. The console window at the bottom shows the execution of these commands: PS C:\Users\usuario> \$nombre = "Camilo", PS C:\Users\usuario> \$saludo = "Hola", PS C:\Users\usuario> \$mensaje = "\$saludo \$nombre", PS C:\Users\usuario> Write-Host \$mensaje, and the output "Hola Camilo" (with a red arrow pointing to the output). The status bar at the bottom indicates "Completado" and "Lin. 1 Col. 1".

```
Sin título1.ps1 X
1 $nombre = "Camilo"
2 $saludo = "Hola"
3 $mensaje = "$saludo $nombre"
4 Write-Host $mensaje

PS C:\Users\usuario>
$nombre = "Camilo"
$saludo = "Hola"
$mensaje = "$saludo $nombre"
Write-Host $mensaje
Hola Camilo

PS C:\Users\usuario>
```

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)
```

o incluso:

```
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:

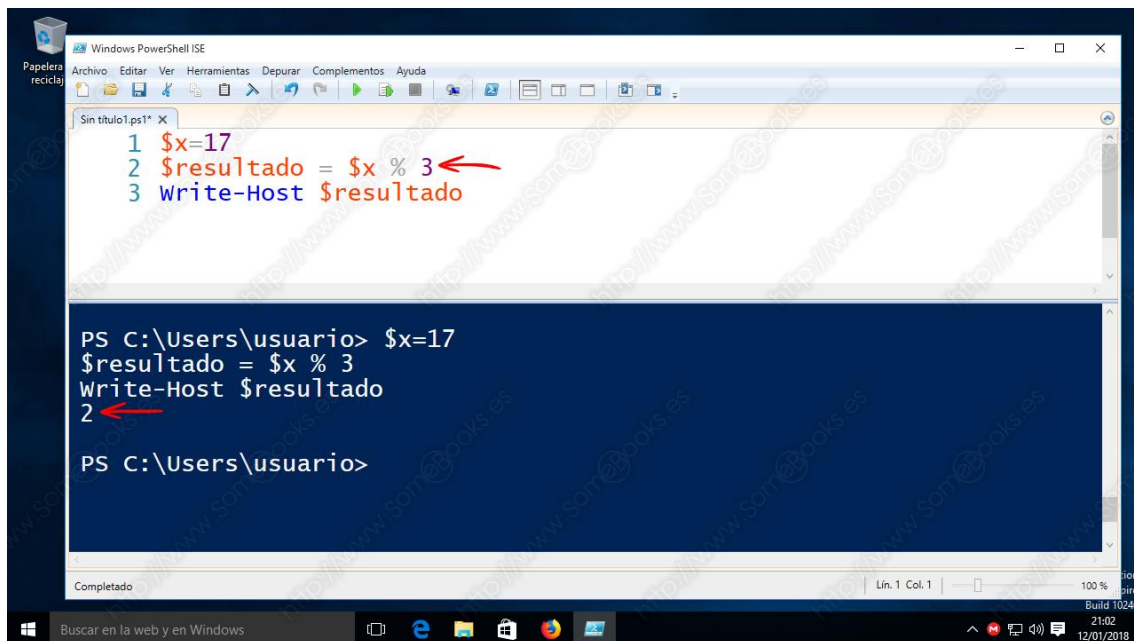
Operadores aritméticos		
Operador	Función	Ejemplo
+	suma	\$resultado = \$x + 3
-	resta	\$resultado = \$x - 3
*	multiplicación	\$resultado = \$x * 3
/	división	\$resultado = \$x / 3
%	resto	\$resultado = \$x % 3

De ellos, probablemente el único que necesite una explicación 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.

Veamos un ejemplo:

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

Y cuando lo ejecutemos, obtendremos algo así:



```
Sin título1.ps1 X
1 $x=17
2 $resultado = $x % 3
3 Write-Host $resultado

PS C:\Users\usuario> $x=17
$resultado = $x % 3
Write-Host $resultado
2

PS C:\Users\usuario>
```

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 3 unidades el valor contenido en la variable $\$x$, podríamos escribir la expresión:

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

Pero también podríamos resumirlo escribiendo:

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

Esto podríamos aplicarlo a todos los operadores aritméticos:

Operadores aritméticos especiales			
Operador	Significado	Ejemplo	Equivalencia
<code>+=</code>	Incrementa el valor de la variable	<code>\$x += 3</code>	<code>\$x = \$x + 3</code>
<code>-=</code>	Reduce el valor de la variable	<code>\$x -= 3</code>	<code>\$x = \$x - 3</code>
<code>*=</code>	Multiplica el valor de la variable	<code>\$x *= 3</code>	<code>\$x = \$x * 3</code>
<code>/=</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. Me refiero al caso en que debamos incrementar o decrementar el contenido de la variable en uno. En estos casos, en lugar de escribir:

`$x = $x + 1`

o escribir:

`$x += 1`

podríamos escribir, sencillamente:

`$x++`

... Y lo mismo podemos hacer con la resta:

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

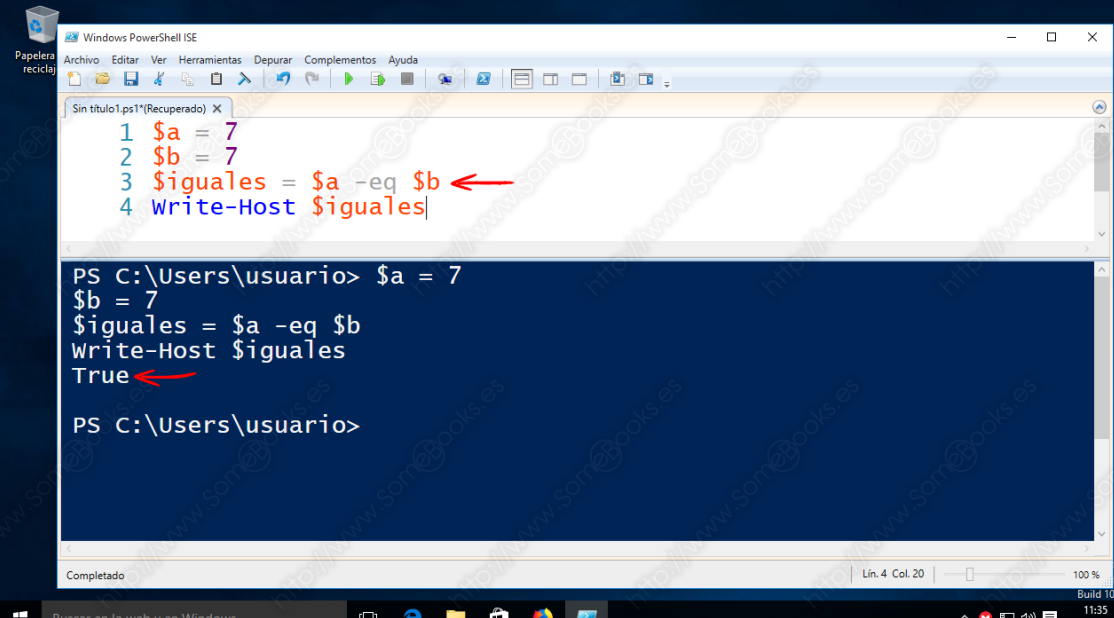
Para asignar el valor de forma directa, basta con escribir algo como esto:

```
$salir = $false
```

Sin embargo, para crear una expresión lógica, necesitaremos una serie de operadores que no hemos visto hasta ahora: 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**.



The screenshot shows the Windows PowerShell ISE interface. The script editor at the top contains four lines of code: `1 $a = 7`, `2 $b = 7`, `3 $iguales = $a -eq $b`, and `4 Write-Host $iguales`. A red arrow points to the `$b` variable in the third line. The console window below shows the execution output: `PS C:\Users\usuario> $a = 7`, `$b = 7`, `$iguales = $a -eq $b`, `Write-Host $iguales`, and `True`. A red arrow points to the output `True`. The status bar at the bottom indicates 'Completado' and 'Lin. 4 Col. 20'.

```
Sin título1.ps1*(Recuperado) X
1 $a = 7
2 $b = 7
3 $iguales = $a -eq $b
4 Write-Host $iguales

PS C:\Users\usuario> $a = 7
$b = 7
$iguales = $a -eq $b
Write-Host $iguales
True

PS C:\Users\usuario>
```

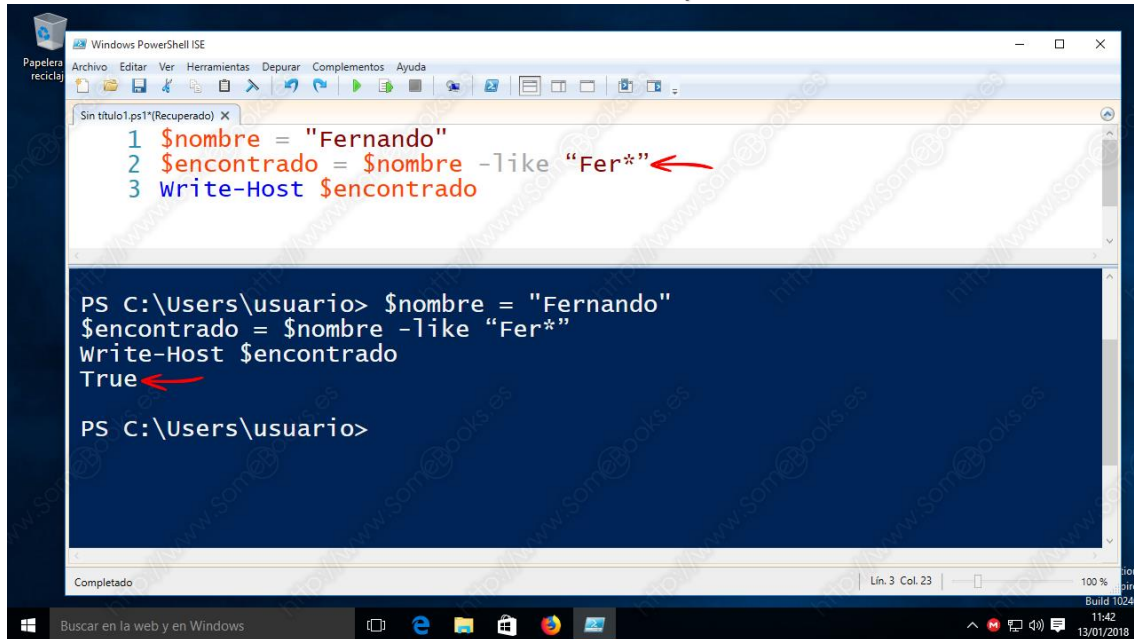

Operadores de comparación		
Operador de comparación	Significado	Ejemplo (\$true)
-eq	Igual	1 -eq 1
-ieq	Igual Con valores de tipo texto, no tendrá en cuenta la diferencia entre mayúsculas y minúsculas	"Hola" -ieq "HOLA"
-ceq	Igual Con valores de tipo texto, tendrá en cuenta la diferencia entre mayúsculas y minúsculas	"HOLA" -ceq "HOLA"
-ne	Diferente	3 -ne 5
-lt	Menor	3 -lt 5
-le	Menor o igual	5 -le 5 3 -le 5
-gt	Mayor	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*"

Como puedes suponer, la variable **\$encontrado** recibe el valor **\$true** sólo

si el contenido de la variable nombre comienza por las letras *Fer*.



```
Sin titulo1.ps1(Recuperado) X
1 $nombre = "Fernando"
2 $encontrado = $nombre -like "Fer*"
3 Write-Host $encontrado

PS C:\Users\usuario> $nombre = "Fernando"
$encontrado = $nombre -like "Fer*"
Write-Host $encontrado
True
PS C:\Users\usuario>
```

Los operadores de este tipo son:

Operadores de comparación		
Operador de comparación	Significado	Ejemplo (\$true)
-like	Es como	"Fermín" -like "Fer*"
-notlike	No es como	"Fermín" -notlike "Ferm*"
-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 así:

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

En este caso, la variable **\$mayor** recibirá el valor **\$true** sólo cuando el valor de **\$a** sea superior al de **\$b** y **\$c**. Veamos un ejemplo:

```

Sin título1.ps1(Recuperado) X
1 $a = 7
2 $b = 5
3 $c = 3
4 $mayor = ($a -gt $b) -and ($a -gt $c)
5 Write-Host $mayor

PS C:\Users\usuario> $a = 7
$b = 5
$c = 3
$mayor = ($a -gt $b) -and ($a -gt $c)
Write-Host $mayor
True
PS C:\Users\usuario>
  
```

En **PowerShell** tenemos los siguientes operadores lógicos:

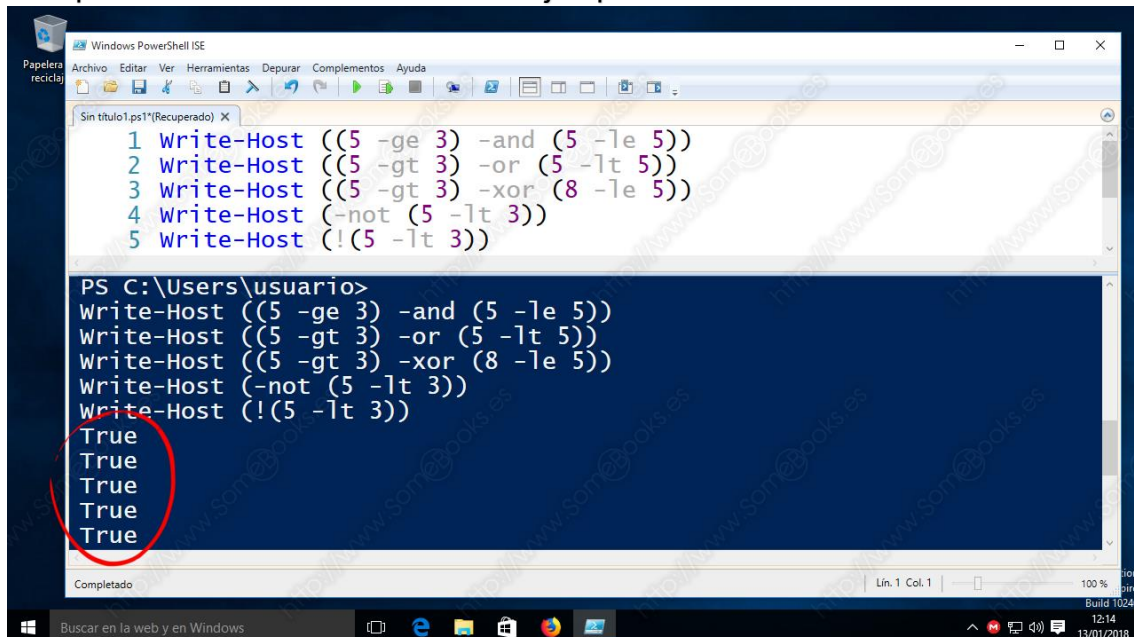
Operadores lógicos		
Operador de comparación	Descripción	Ejemplo (\$true)
-and	Devuelve \$true cuando las dos expresiones que intervienen ofrecen el valor \$true	(5 -ge 3) -and (5 -le 5)
-or	Devuelve \$true cuando alguna de las expresiones que intervienen ofrecen el valor \$true.	(5 -gt 3) -or (5 -lt 5)
-xor	Devuelve \$true cuando sólo una de las expresiones que intervienen ofrecen el valor \$true.	(5 -gt 3) -xor (8 -le 5)
-not !	Devuelve \$true cuando la expresión sobre la que actúa ofrecen el valor \$false.	-not (5 -lt 3) !(5 -lt 3)

Observa que 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.

Comprobemos los resultados de los ejemplos anteriores:



```
Sin título1.ps1*(Recuperado) X
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 3))

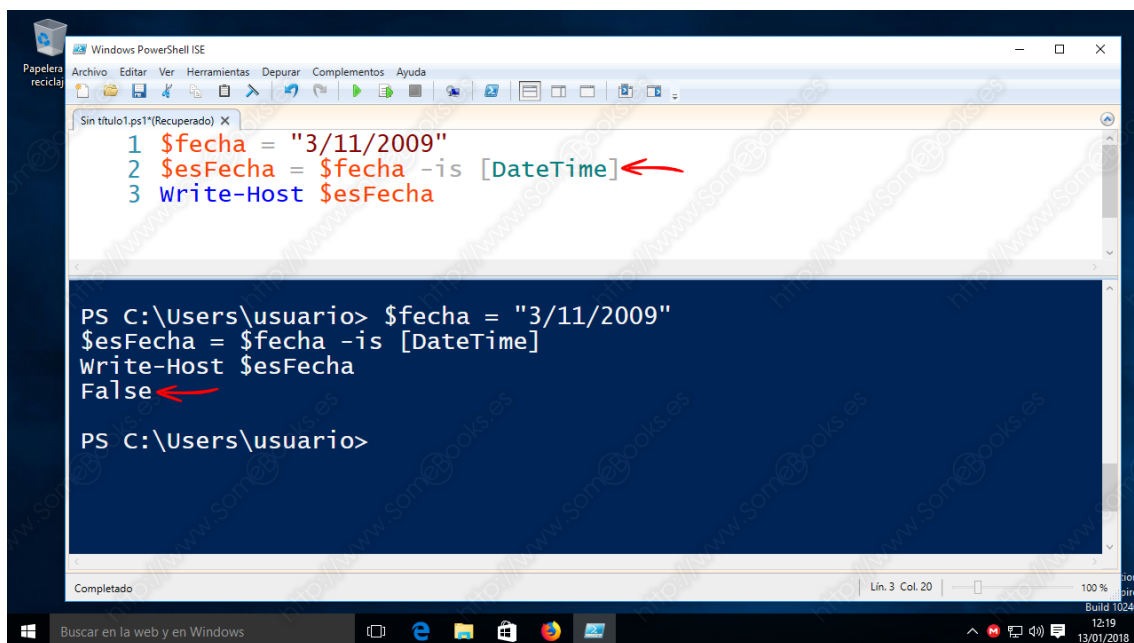
PS C:\Users\usuario>
Write-Host ((5 -ge 3) -and (5 -le 5))
Write-Host ((5 -gt 3) -or (5 -lt 5))
Write-Host ((5 -gt 3) -xor (8 -le 5))
Write-Host (-not (5 -lt 3))
Write-Host (!(5 -lt 3))
True
True
True
True
True
```

Operadores de tipo

Permiten comprobar si una variable (o cualquier otro valor) se corresponde con un tipo de dato en particular. Veamos un ejemplo:

```
$fecha = "3/11/2009"
$esFecha = $fecha -is [DateTime]
```

En este caso, la variable **\$esFecha** recibe el valor **\$false**, porque el dato, aunque parece una fecha, en realidad ya sabemos que es un texto (su tipo de dato es *string*).



```
Sin titulo1.ps1*(Recuperado) X
1 $fecha = "3/11/2009"
2 $esFecha = $fecha -is [DateTime]
3 Write-Host $esFecha

PS C:\Users\usuario> $fecha = "3/11/2009"
$esFecha = $fecha -is [DateTime]
Write-Host $esFecha
False

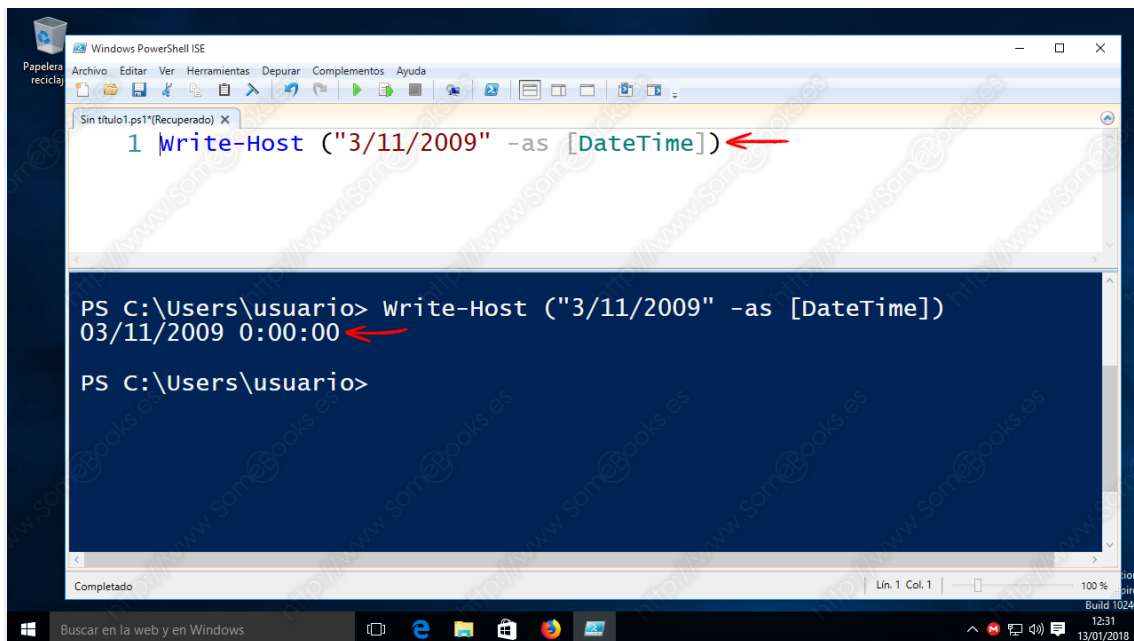
PS C:\Users\usuario>
```

Operadores lógicos		
Operador de comparación	Descripción	Ejemplo (\$true)
-is	Devuelve \$true cuando el dato se corresponde con el tipo indicado	"3/11/2009" -is [string]
-isnot	Devuelve \$true cuando el dato no se corresponde con el tipo indicado	"3/11/2009" -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 ("3/11/2009" -as [DateTime])
```

El resultado sería este:



```
Sin título1.ps1*(Recuperado) X
1 Write-Host ("3/11/2009" -as [DateTime])

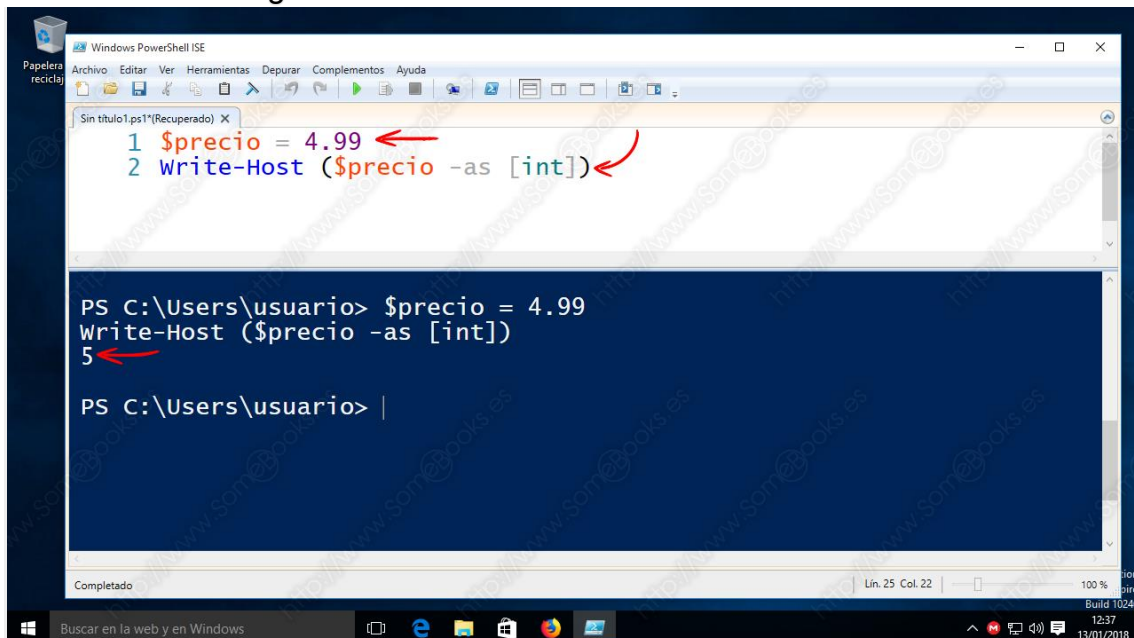
PS C:\Users\usuario> Write-Host ("3/11/2009" -as [DateTime])
03/11/2009 0:00:00

PS C:\Users\usuario>
```

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. Por ejemplo, cuando escribimos lo siguiente:

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

Obtendremos el siguiente resultado:



```
Sin título1.ps1*(Recuperado) X
1 $precio = 4.99
2 Write-Host ($precio -as [int])

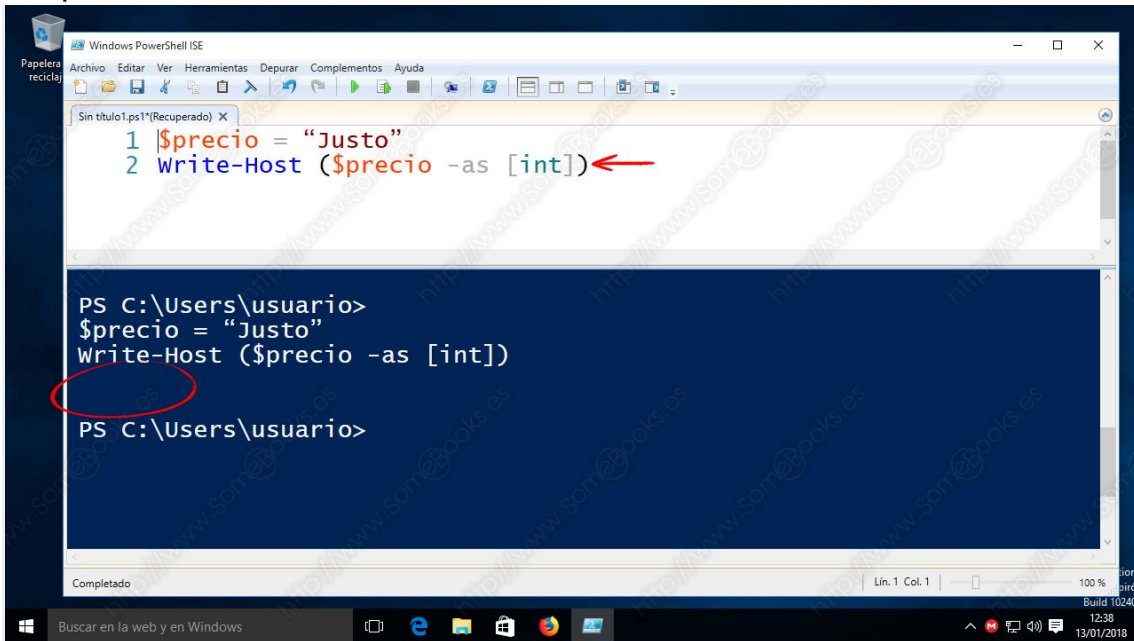
PS C:\Users\usuario> $precio = 4.99
Write-Host ($precio -as [int])
5

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

Sin embargo, cuando esto no sea posible, sencillamente no hará nada. Por ejemplo, si escribimos lo siguiente:

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

Lo que obtenemos es esto:



```
Sin título1.ps1*(Recuperado) X
1 $precio = "Justo"
2 Write-Host ($precio -as [int])

PS C:\Users\usuario>
$precio = "Justo"
Write-Host ($precio -as [int])
PS C:\Users\usuario>
```

Variables como objetos

Ya dijimos al principio de esta documentación que *PowerShell* está basado en *.NET Framework*. En la práctica, esto significa que 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.

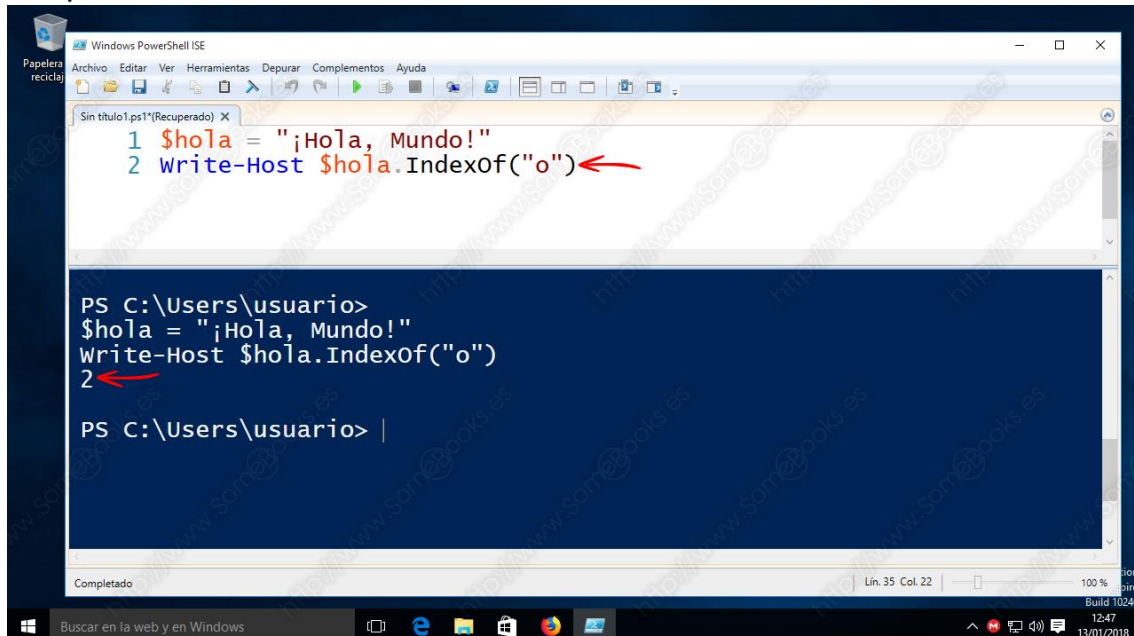
Nuestro objetivo no es profundizar aquí sobre conceptos de programación orientada a objetos, pero sí te mostraremos cómo usar algunos métodos, asociados a diferentes tipos de datos, para resolver algunas tareas concretas. Veamos un ejemplo: Imagina que tienes la siguiente variable y necesitas averiguar la posición en la que aparece la primera letra 'o'.

```
$holaa = "¡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")
```

Y aquí vemos el resultado:



```
Sin título1.ps1*(Recuperado) X
1 $hola = "¡Hola, Mundo!"
2 Write-Host $hola.IndexOf("o")

PS C:\Users\usuario>
$hola = "¡Hola, Mundo!"
Write-Host $hola.IndexOf("o")
2

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

Como ves, el resultado es 2 (y no tres, como cabía esperar). El motivo es que las posiciones de los caracteres se cuentan a partir de cero.

Otro ejemplo: obtener el contenido de la variable anterior, pero convertido en minúsculas:

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

Y vamos a comprobar el resultado:

The screenshot shows the Windows PowerShell ISE interface. The script editor at the top contains two lines of code: `1 $hoLa = "¡HoLa, Mundo!"` and `2 Write-Host $hoLa.ToLower()`. A red arrow points to the `ToLower()` method. The console window below shows the execution of these commands, resulting in the output `¡hoLa, mundo!`. Another red arrow points to this output. The status bar at the bottom indicates the script is 'Completado' (Completed).

En realidad, cada tipo de datos tiene multitud de métodos a los que podemos recurrir. Tantos que sería demasiado extenso abordarlos aquí. No obstante, a lo largo de esta publicación iremos incluyendo diversos ejemplos de su uso. Si necesitas consultar la lista completa de métodos disponibles para cada tipo de dato, te recomiendo echar un vistazo a los siguientes enlaces:

- String: [https://msdn.microsoft.com/en-us/library/system.string\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.string(v=vs.110).aspx#Methods)
- Char: [https://msdn.microsoft.com/en-us/library/system.char\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.char(v=vs.110).aspx#Methods)
- Byte: [https://msdn.microsoft.com/en-us/library/system.byte\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.byte(v=vs.110).aspx#Methods)
- Int (Int32): [https://msdn.microsoft.com/en-us/library/system.int32\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.int32(v=vs.110).aspx#Methods)
- Long (int64): [https://msdn.microsoft.com/en-us/library/system.int64\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.int64(v=vs.110).aspx#Methods)
- Single: [https://msdn.microsoft.com/en-us/library/system.single\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.single(v=vs.110).aspx#Methods)
- Double: [https://msdn.microsoft.com/en-us/library/system.double\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.double(v=vs.110).aspx#Methods)
- Decimal: [https://msdn.microsoft.com/en-us/library/system.decimal\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.decimal(v=vs.110).aspx#Methods)

- DateTime: [https://msdn.microsoft.com/en-us/library/system.datetime\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.datetime(v=vs.110).aspx#Methods)
- Bool (Boolean): [https://msdn.microsoft.com/en-us/library/system.boolean\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.boolean(v=vs.110).aspx#Methods)
- Array (ArrayList): [https://msdn.microsoft.com/es-es/library/system.collections.arraylist\(v=vs.110\).aspx#M%C3%A9todos](https://msdn.microsoft.com/es-es/library/system.collections.arraylist(v=vs.110).aspx#M%C3%A9todos)
- Hashtable: [https://msdn.microsoft.com/es-es/library/system.collections.hashtable\(v=vs.110\).aspx#M%C3%A9todos](https://msdn.microsoft.com/es-es/library/system.collections.hashtable(v=vs.110).aspx#M%C3%A9todos)
- Xml: [https://msdn.microsoft.com/en-us/library/system.xml.xmldocument\(v=vs.110\).aspx#Methods](https://msdn.microsoft.com/en-us/library/system.xml.xmldocument(v=vs.110).aspx#Methods)

Punto 5: Estructuras condicionales

Introducción

Todos los lenguajes de programación procedimentales necesitan estructuras que faciliten la ejecución de determinadas instrucciones sólo cuando se cumpla una condición concreta. Por ejemplo, un *script* puede pedir un dato al usuario y, dependiendo de la respuesta, ejecutar un fragmento de código u otro. En definitiva, la lógica condicional permite a los *scripts* tomar decisiones, añadiéndoles flexibilidad e inteligencia a su estructura.

Como la mayoría de los lenguajes de programación, *PowerShell* utiliza para estos fines dos instrucciones diferentes:

1. *if*: Examina un valor lógico y, según sea el resultado, ejecutará un conjunto de instrucciones u otro.
2. *switch*: Su funcionamiento es parecido al anterior, pero nos permite evaluar más de una condición.

Uso de la orden if

Aunque la orden *if* puede tener una sintaxis más compleja, comenzaremos por su forma más sencilla. En ese caso, su formato es así:

```
if (condición) {Bloque de código}
```

Como habrás imaginado, el significado de sus elementos es este:

- La palabra *condición* se refiere a una expresión lógica. Es decir, que al evaluarla se obtendrá un valor **\$true** o **\$false**.
- Por su parte, el bloque de código será un conjunto de instrucciones que sólo se ejecutarán cuando la condición ofrezca el valor **\$true**. El bloque de código se escribe siempre encerrado en una pareja de llaves.

Veamos un ejemplo:


```
[int]$numero = [int] (Read-Host "Escribe un número")
[bool]$positivo = ($numero -ge 0)
if ($positivo) {Write-Host "El número es positivo"}
```

El resultado que obtendríamos sería parecido a este:

The screenshot shows the Windows PowerShell ISE interface. The script editor contains three lines of code: `[int]$numero = [int] (Read-Host "Escribe un número")`, `[bool]$positivo = ($numero -ge 0)`, and `if ($positivo) {Write-Host "El número es positivo"}`. The console output shows the prompt `PS C:\Users\usuario>`, followed by the command execution, then the prompt `Escribe un número: 8` (with a red arrow pointing to the input), followed by the output `El número es positivo` (with a red arrow pointing to the output), and finally the prompt `PS C:\Users\usuario>`. The status bar at the bottom indicates 'Completado', 'Lin. 3 Col. 52', and '100 %'.

En el ejemplo anterior, hemos utilizado una variable lógica para evaluar la expresión. Sin embargo, podríamos haber abreviado escribiendo la expresión directamente en la orden *if*. Así, nos ahorraríamos usar la variable `$positivo`:

```
[int]$numero = [int] (Read-Host "Escribe un número")
if ($numero -ge 0) {Write-Host "El número es positivo"}
```

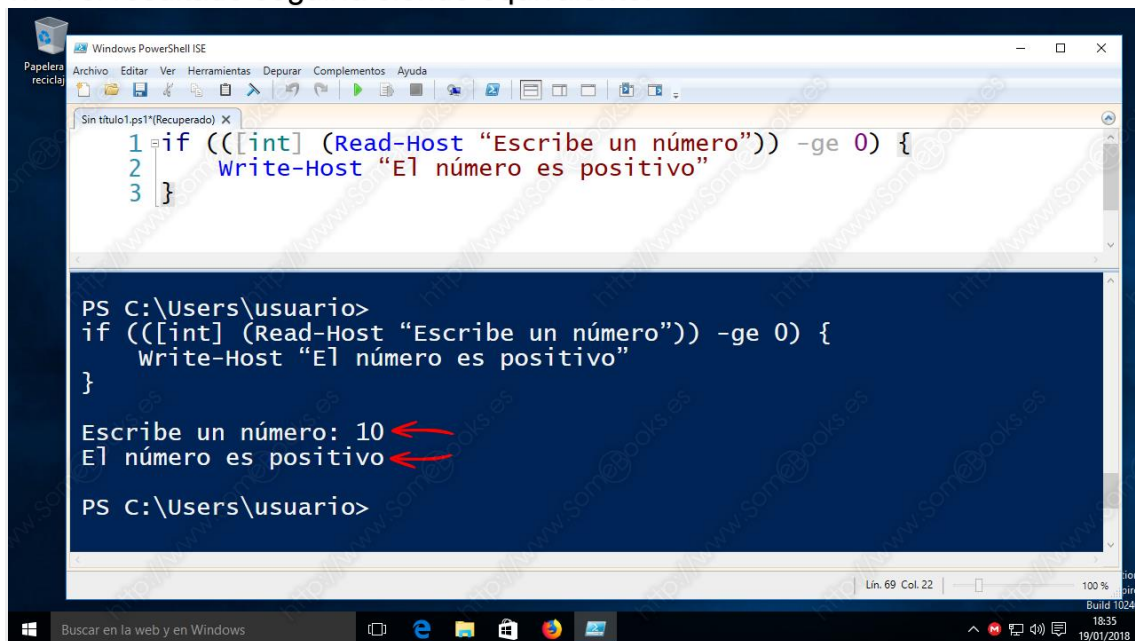
... Y el resultado sería similar:

The screenshot shows the Windows PowerShell ISE interface. The script editor contains two lines of code: `[int]$numero = [int] (Read-Host "Escribe un número")` and `if ($numero -ge 0) {Write-Host "El número es positivo"}`. The console output shows the prompt `PS C:\Users\usuario>`, followed by the command execution, then the prompt `Escribe un número: 6` (with a red arrow pointing to the input), followed by the output `El número es positivo` (with a red arrow pointing to the output), and finally the prompt `PS C:\Users\usuario>`. The status bar at the bottom indicates 'Completado', 'Lin. 30 Col. 22', and '100 %'.

De hecho, dentro del paréntesis podemos utilizar cualquier expresión válida que ofrezca un valor lógico. Además, hemos aprovechado para escribir el *if* de una forma más bonita...

```
if ([[int] (Read-Host "Escribe un número"))) -ge 0) {  
    Write-Host "El número es positivo"  
}
```

... Y el resultado seguiría siendo equivalente:



```
Sin título1.ps1*(Recuperado) X  
1 if ([[int] (Read-Host "Escribe un número"))) -ge 0) {  
2     Write-Host "El número es positivo"  
3 }  
  
PS C:\Users\usuario>  
if ([[int] (Read-Host "Escribe un número"))) -ge 0) {  
    Write-Host "El número es positivo"  
}  
  
Escribe un número: 10  
El número es positivo  
  
PS C:\Users\usuario>
```

Esto nos ahorraría también la variable *\$numero*... Pero quizás sea mejor mantener el código más legible.

En definitiva, todos los ejemplos anteriores son equivalentes. Cuando los ejecutes, si escribes un número positivo, el *script* te contestará con el texto *El número es positivo*. Y si escribes un número negativo, no aparecerá nada.

Uso de la orden *if* con *else*

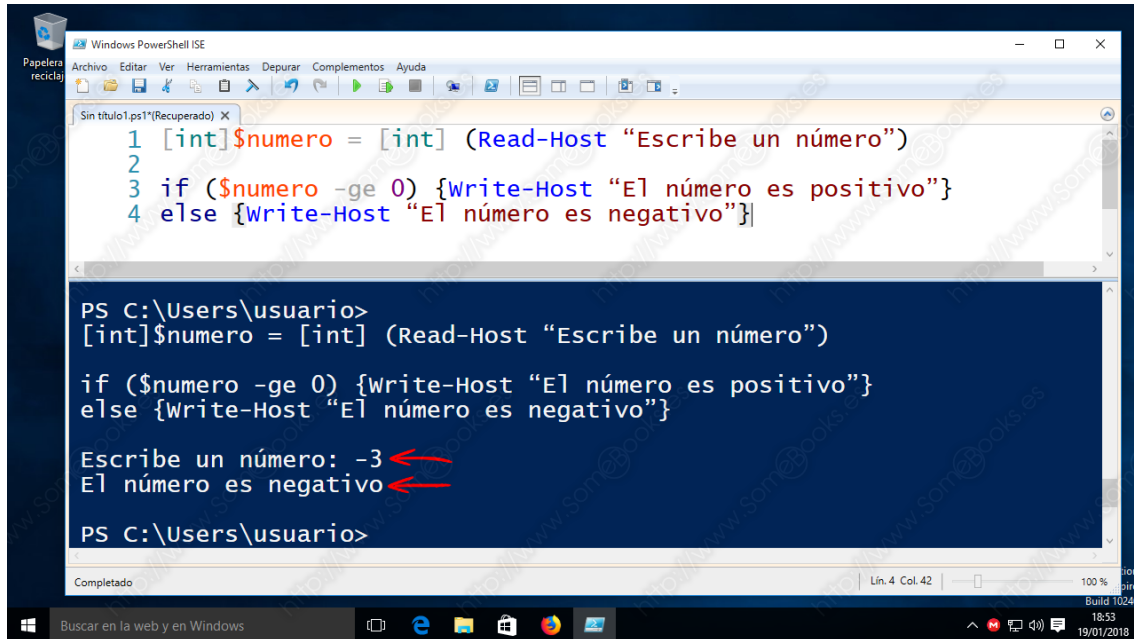
Añadiendo *else* a la orden *if*, podremos incluir un bloque de código que sólo se ejecute cuando la condición sea falsa. En ese caso, su formato es así:

```
if (condición) {Bloque de código 1}  
else {Bloque de código 2}
```

Veamos un ejemplo:

```
[int]$numero = [int] (Read-Host "Escribe un número")  
  
if ($numero -ge 0) {Write-Host "El número es positivo"}  
else {Write-Host "El número es negativo"}
```

El resultado obtenido sería este:



```
Sin titulo1.ps1*(Recuperado) X
1 [int]$numero = [int] (Read-Host "Escribe un número")
2
3 if ($numero -ge 0) {Write-Host "El número es positivo"}
4 else {Write-Host "El número es negativo"}

PS C:\Users\usuario>
[int]$numero = [int] (Read-Host "Escribe un número")

if ($numero -ge 0) {Write-Host "El número es positivo"}
else {Write-Host "El número es negativo"}

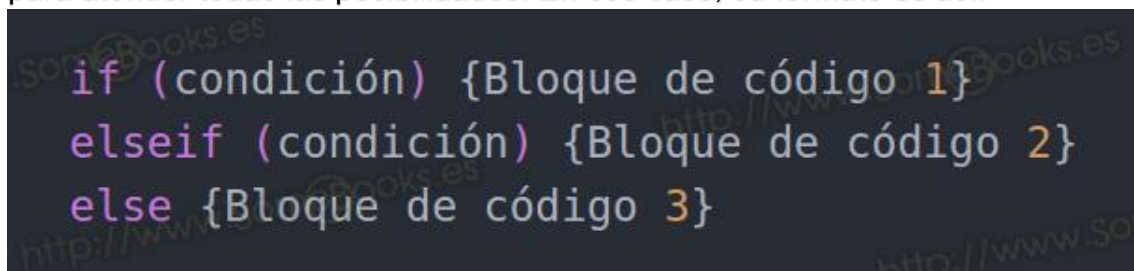
Escribe un número: -3
El número es negativo

PS C:\Users\usuario>
```

Como ves, ahora el *script* siempre contesta, indicándonos si el número que hemos escrito es positivo o negativo.

Uso de la orden `if` con `elseif`

En situaciones donde tenemos más de dos alternativas, podemos incluir `elseif` para atender todas las posibilidades. En ese caso, su formato es así:



```
if (condición) {Bloque de código 1}
elseif (condición) {Bloque de código 2}
else {Bloque de código 3}
```

En este caso, puedes utilizar tantos *elseif* como requiera el problema que estés resolviendo.

Veamos un ejemplo: Imagina que necesitas obtener una respuesta del usuario y debes tener en cuenta que éste no escriba ninguna de las respuestas esperadas. Podríamos hacerlo así:

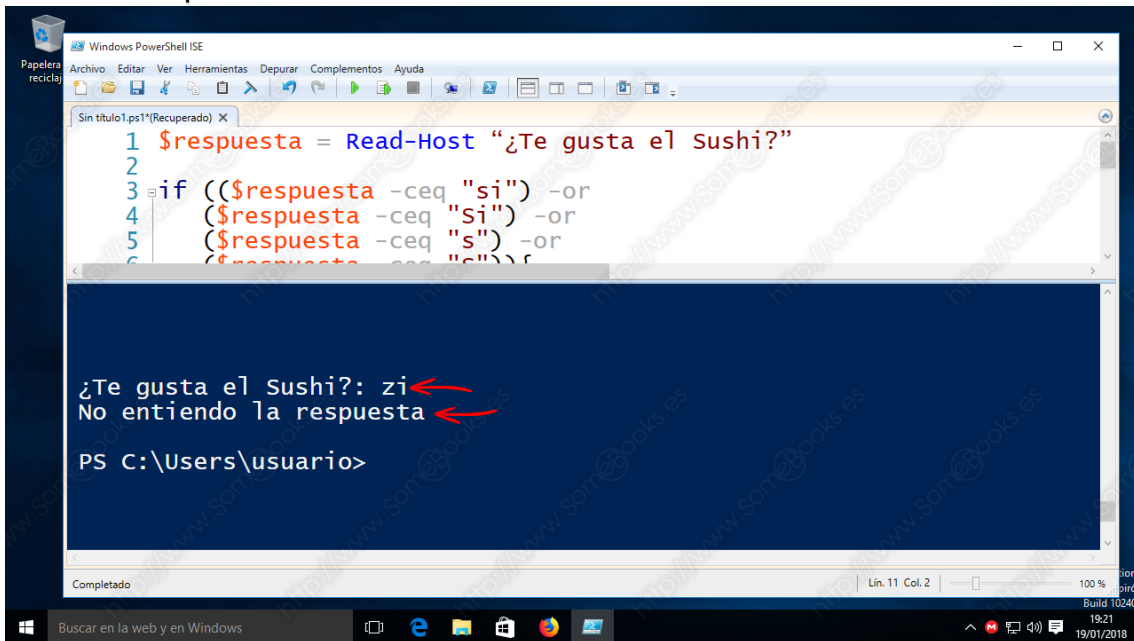
```

$respuesta = Read-Host "¿Te gusta el sushi?"

if (($respuesta -ceq "si") -or
    ($respuesta -ceq "Si") -or
    ($respuesta -ceq "s") -or
    ($respuesta -ceq "S")){
    Write-Host "Tu respuesta ha sido afirmativa"
}
elseif (($respuesta -ceq "no") -or
    ($respuesta -ceq "No") -or
    ($respuesta -ceq "n") -or
    ($respuesta -ceq "N")){
    Write-Host "Tu respuesta ha sido negativa"
}
else{
    Write-Host "No entiendo la respuesta"
}

```

El resultado podría ser como este:



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

```

Sin título1.ps1*(Recuperado) X
1 $respuesta = Read-Host "¿Te gusta el sushi?"
2
3 if (($respuesta -ceq "si") -or
4     ($respuesta -ceq "Si") -or
5     ($respuesta -ceq "s") -or
6     ($respuesta -ceq "S")){
7     Write-Host "Tu respuesta ha sido afirmativa"
8 }
9 elseif (($respuesta -ceq "no") -or
10     ($respuesta -ceq "No") -or
11     ($respuesta -ceq "n") -or
12     ($respuesta -ceq "N")){
13     Write-Host "Tu respuesta ha sido negativa"
14 }
15 else{
16     Write-Host "No entiendo la respuesta"
17 }

```

The output of the script is displayed in the console window:

```

¿Te gusta el sushi?: zi
No entiendo la respuesta
PS C:\Users\usuario>

```

Red arrows indicate that the input 'zi' did not match any of the conditions in the script, leading to the 'else' block's output: 'No entiendo la respuesta'.

En realidad, podríamos haber simplificado las condiciones si no diferenciáramos entre minúsculas y mayúsculas:


```

$respuesta = Read-Host "¿Te gusta el Sushi?"

if (($respuesta -eq "si") -or ($respuesta -eq "s")){
    Write-Host "Tu respuesta ha sido afirmativa"
}
elseif (($respuesta -eq "no") -or ($respuesta -eq "n")){
    Write-Host "Tu respuesta ha sido negativa"
}
else{
    Write-Host "No entiendo la respuesta"
}

```

El inconveniente es que, ahora, pueden darse respuestas que no habíamos tenido en cuenta al principio. Por ejemplo, *Sl* ó *sI*.

Otra opción sería utilizar el método *IndexOf* que ya conocemos del Punto anterior:

```

$afirmativo = "si Si sí Sí s "
$negativo = "no No n "

$respuesta = Read-Host "¿Te gusta el Sushi?"

if ($afirmativo.IndexOf($respuesta+" ") -ge 0){
    Write-Host "Tu respuesta ha sido afirmativa"
}
elseif ($negativo.IndexOf($respuesta+" ") -ge 0){
    Write-Host "Tu respuesta ha sido negativa"
}
else{
    Write-Host "No entiendo la respuesta"
}

```

La ventaja de este método es que, si queremos añadir nuevas opciones de respuesta afirmativa o negativa, sólo tenemos que modificar las variables correspondientes, sin complicar la lógica del programa.

Uso de la orden switch

La orden *switch* es una alternativa a *if* para los casos en los que tenemos una gran cantidad de opciones.


```
Switch (valor de prueba) {  
  Patrón 1 {Bloque de código 1}  
  Patrón 2 {Bloque de código 2}  
  Patrón n {Bloque de código n}  
  default {Bloque por defecto }  
}
```

Dentro del paréntesis que acompaña a la orden *switch* deberemos incluir una variable o una expresión. A continuación, incluiremos los posibles resultados y, junto a cada uno de ellos, el bloque de instrucciones que deben ejecutarse si dicho valor coincide con el resultado de evaluar la variable o expresión de *switch*.

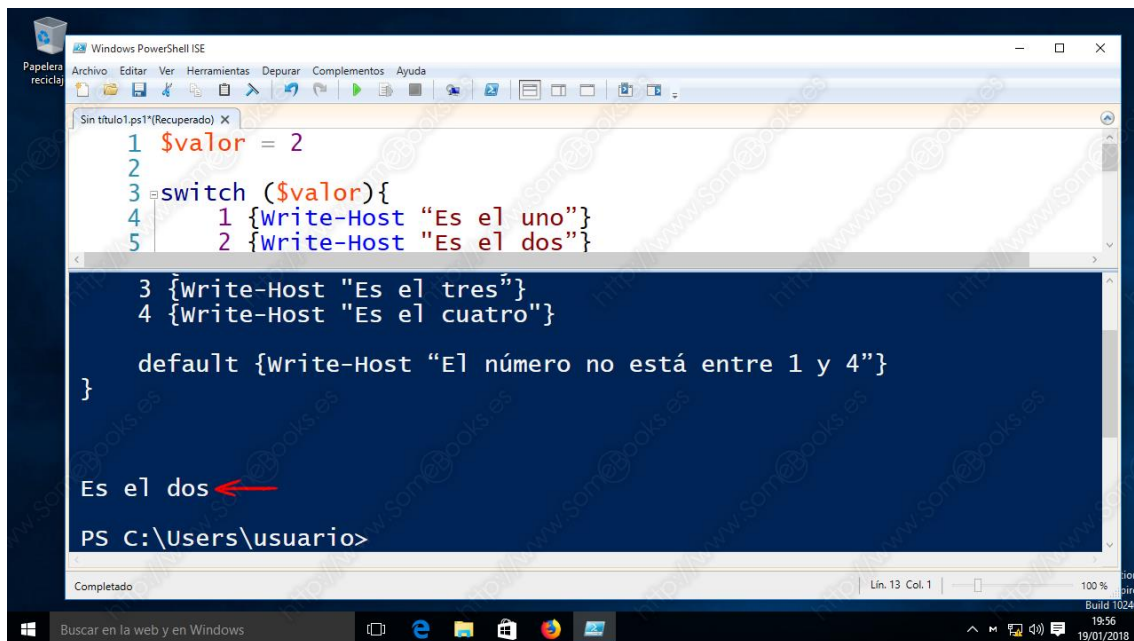
Si no coincide ninguno de los patrones, de forma opcional, podemos incluir la cláusula *default* con un bloque que se ejecutará por defecto.

Observa que, además de la pareja de llaves que engloba cada bloque de código, existe una pareja de llaves que engloba la estructura completa.

Veamos un ejemplo:

```
$valor = 2  
  
switch ($valor){  
  1 {Write-Host "Es el uno"}  
  2 {Write-Host "Es el dos"}  
  3 {Write-Host "Es el tres"}  
  4 {Write-Host "Es el cuatro"}  
  
  default {Write-Host "El número no está entre 1 y 4"}  
}
```

El resultado sería este:



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

```
1 $valor = 2
2
3 switch ($valor){
4     1 {Write-Host "Es el uno"}
5     2 {Write-Host "Es el dos"}
6
7     3 {Write-Host "Es el tres"}
8     4 {Write-Host "Es el cuatro"}
9
10    default {Write-Host "El número no está entre 1 y 4"}
11 }

```

The console window shows the output of the script:

```
Es el dos
PS C:\Users\usuario>

```

A red arrow points to the output "Es el dos". The status bar at the bottom indicates "Completado" (Completed).

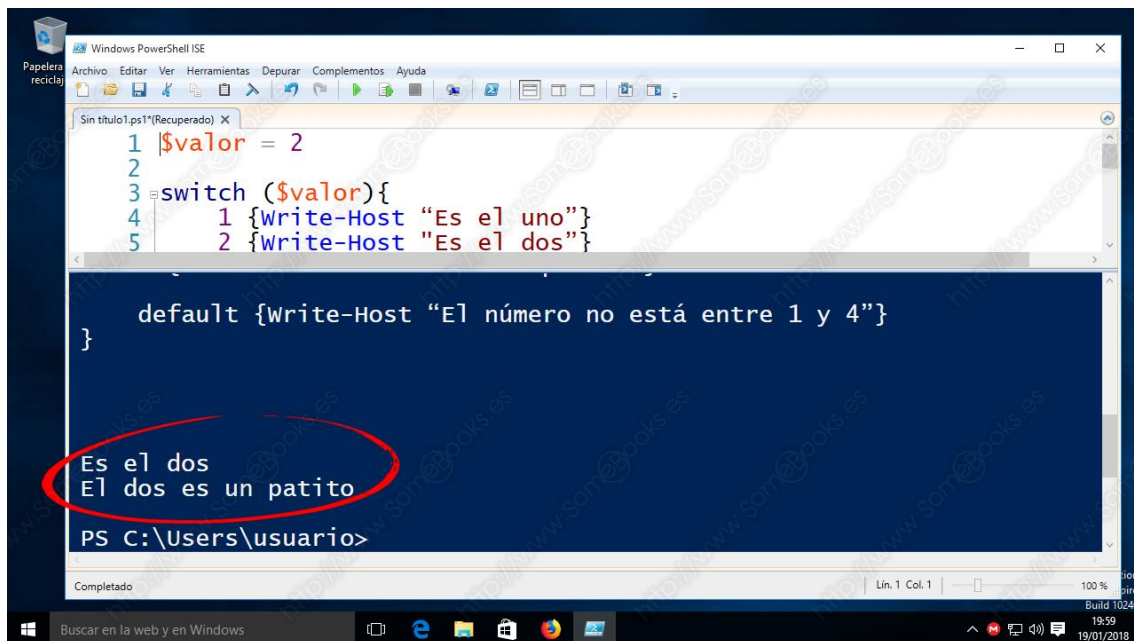
Una de las características que diferencian a *switch* de *if* es que, en este caso, siempre se evalúan todos los patrones. Por ejemplo, observa el siguiente ejemplo en el que hemos repetido el valor 2:

```
$valor = 2

switch ($valor){
    1 {Write-Host "Es el uno"}
    2 {Write-Host "Es el dos"}
    3 {Write-Host "Es el tres"}
    4 {Write-Host "Es el cuatro"}
    2 {Write-Host "El dos es un patito"}

    default {Write-Host "El número no está entre 1 y 4"}
}
```

Comprobamos que las condiciones no dejan de evaluarse después de la primera coincidencia:



```
Sin título1.ps1 (Recuperado) X
1 $valor = 2
2
3 switch ($valor){
4     1 {Write-Host "Es el uno"}
5     2 {Write-Host "Es el dos"}
6
7     default {Write-Host "El número no está entre 1 y 4"}
8 }
9
10 Es el dos
11 El dos es un patito
12 PS C:\Users\usuario>
```

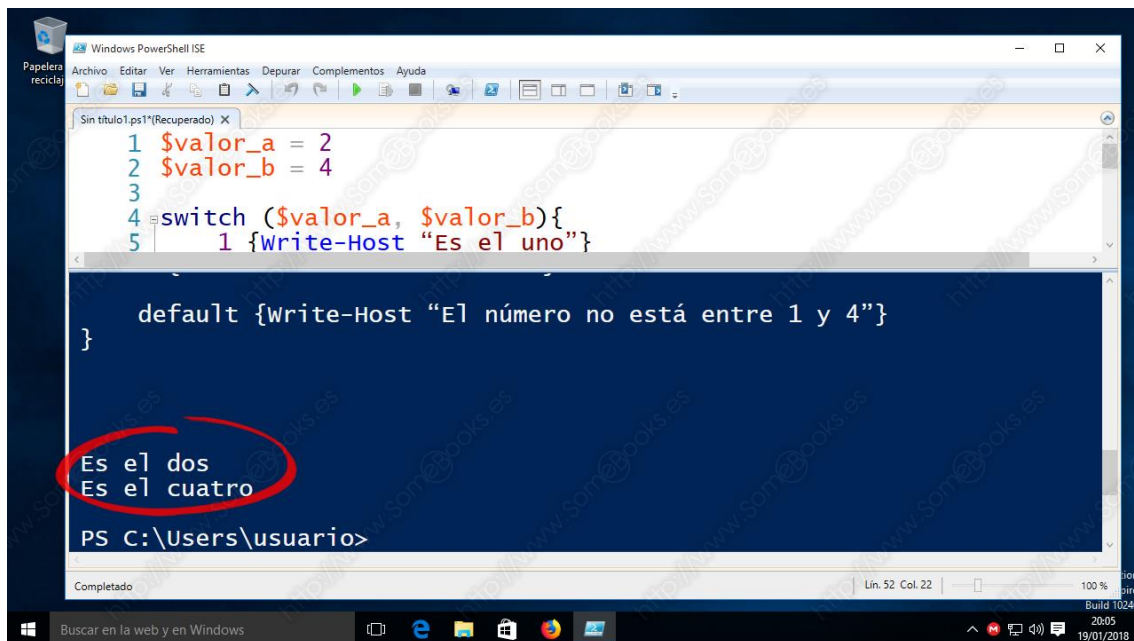
Incluso podemos evaluar diferentes variables o expresiones al mismo tiempo. Sólo tenemos que incluirlas separadas por comas:

```
$valor_a = 2
$valor_b = 4

switch ($valor_a, $valor_b){
    1 {Write-Host "Es el uno"}
    2 {Write-Host "Es el dos"}
    3 {Write-Host "Es el tres"}
    4 {Write-Host "Es el cuatro"}

    default {Write-Host "El número no está entre 1 y 4"}
}
```

Observa que *PowerShell* ha encontrado dos coincidencias (una por cada variable):



```
Sin título1.ps1(Recuperado) X
1 $valor_a = 2
2 $valor_b = 4
3
4 switch ($valor_a, $valor_b){
5     1 {Write-Host "Es el uno"}

    default {Write-Host "El número no está entre 1 y 4"}
}

Es el dos
Es el cuatro

PS C:\Users\usuario>
```

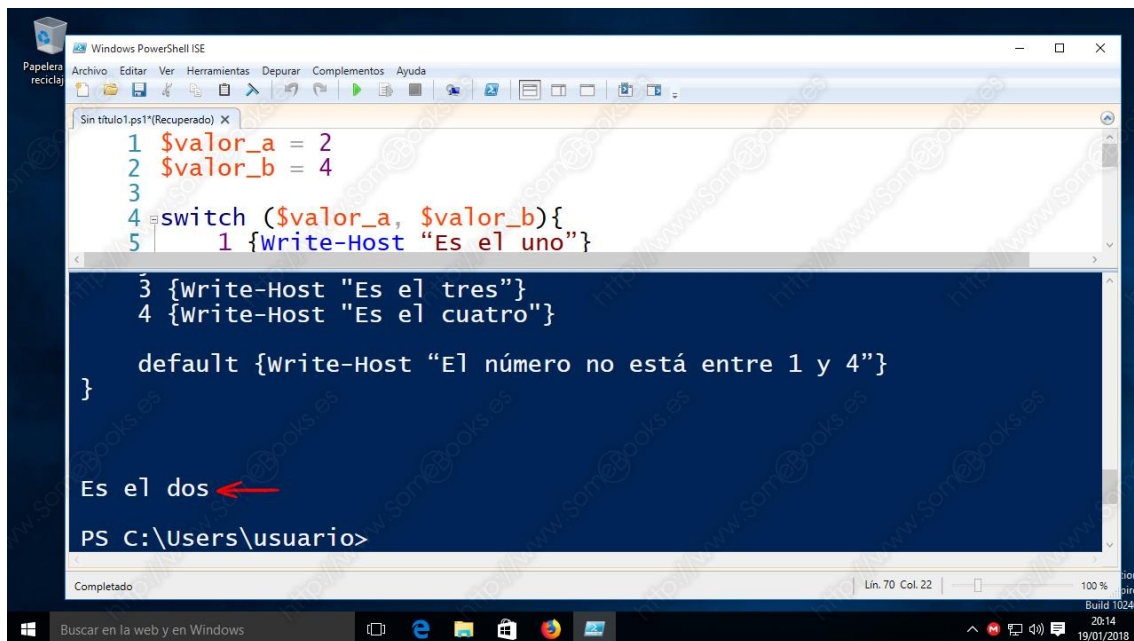
Cuando necesitamos que, tras evaluar un determinado bloque, no se continúe evaluando ningún otro, podemos utilizar la orden *break*. Observa lo que pasa cuando la incluimos en el ejemplo anterior:

```
$valor_a = 2
$valor_b = 4

switch ($valor_a, $valor_b){
    1 {Write-Host "Es el uno"}
    2 {Write-Host "Es el dos"
      break
    }
    3 {Write-Host "Es el tres"}
    4 {Write-Host "Es el cuatro"}

    default {Write-Host "El número no está entre 1 y 4"}
}
```

En el ejemplo, al evaluar el valor 2, ejecuta la orden *break* y se interrumpe la ejecución de *switch*:



```
Sin título1.ps1(Recuperado) X
1 $valor_a = 2
2 $valor_b = 4
3
4 switch ($valor_a, $valor_b){
5     1 {Write-Host "Es el uno"}
6
7     3 {Write-Host "Es el tres"}
8     4 {Write-Host "Es el cuatro"}
9
10    default {Write-Host "El número no está entre 1 y 4"}
11 }
12
13 Es el dos ←
14 PS C:\Users\usuario>
```

Cuando hay más de dos instrucciones en un mismo bloque, lo normal es escribirlas en varias líneas (como hemos hecho en el ejercicio anterior. Sin embargo, también tenemos la opción de escribirlas en la misma línea separándolas con un punto y coma (;):

```
$valor_a = 2
$valor_b = 4

switch ($valor_a, $valor_b){
    1 {Write-Host "Es el uno"}
    2 {Write-Host "Es el dos"; break}
    3 {Write-Host "Es el tres"}
    4 {Write-Host "Es el cuatro"}

    default {Write-Host "El número no está entre 1 y 4"}
}
```

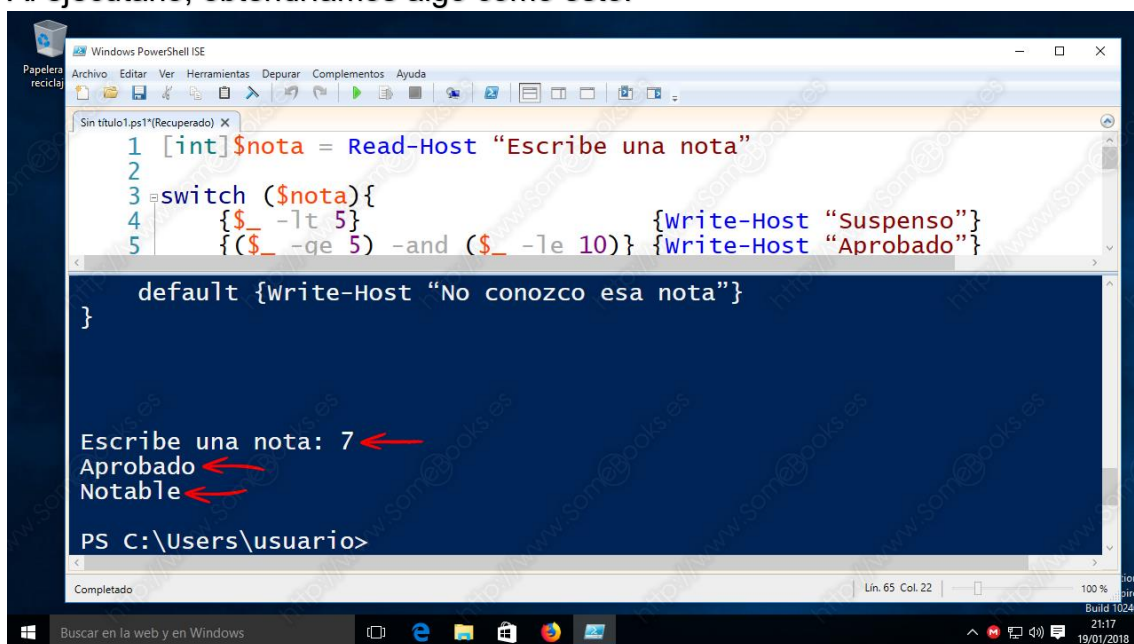
También es frecuente que el valor del patrón pueda obtenerse de una expresión más compleja pero, para que la sintaxis sea correcta, dicha expresión también debe incluirse en una pareja de llaves. En estos casos, el bloque de código se ejecutará cuando el resultado de la expresión sea **\$true**. Veamos un ejemplo:


```
[int]$nota = Read-Host "Escribe una nota:"

switch ($nota){
    {$_ -lt 5} {Write-Host "Suspenso"}
    {($_ -ge 5) -and ($_ -le 10)} {Write-Host "Aprobado"}
    {$_ -in 1..4} {Write-Host "Insuficiente"}
    {$_ -in 5..6} {Write-Host "Suficiente"}
    {$_ -in 7, 8} {Write-Host "Bien"}
    {$_ -in 9, 10} {Write-Host "Notable"}

    default {Write-Host "No conozco esa nota"}}
}
```

Al ejecutarlo, obtendríamos algo como esto:



```
Sin título1.ps1*(Recuperado) X
1 [int]$nota = Read-Host "Escribe una nota"
2
3 switch ($nota){
4     {$_ -lt 5} {Write-Host "Suspenso"}
5     {($_ -ge 5) -and ($_ -le 10)} {Write-Host "Aprobado"}
6
7     default {Write-Host "No conozco esa nota"}}
8 }
9
10 Escribe una nota: 7
11 Aprobado
12 Notable
13
14 PS C:\Users\usuario>
```

Observa que, dentro de la expresión, el símbolo `$_` está representado el resultado de evaluar la variable contenida en `switch`.
Por último, existen algunos modificadores que nos permiten actuar sobre la variable o expresión contenida en el `switch`:

Modificadores de switch	
Parámetro	Descripción
-wildcard	Sólo se aplica a valores de tipo texto e indica que la condición es una cadena comodín.
-exact	Sólo se aplica a valores de tipo texto e indica que debe coincidir exactamente con alguno de los patrones.
-casesensitive	Sólo se aplica a valores de tipo texto e indica que la comparación con el patrón debe coincidir en mayúsculas y minúsculas.
-file	La entrada proviene de un archivo. La orden switch evaluará cada línea del archivo. Si se incluyen varios archivos, sólo se evalúa el último.
-regex	Sólo se aplica a valores de tipo texto y permite utilizar expresiones regulares en la comparación.

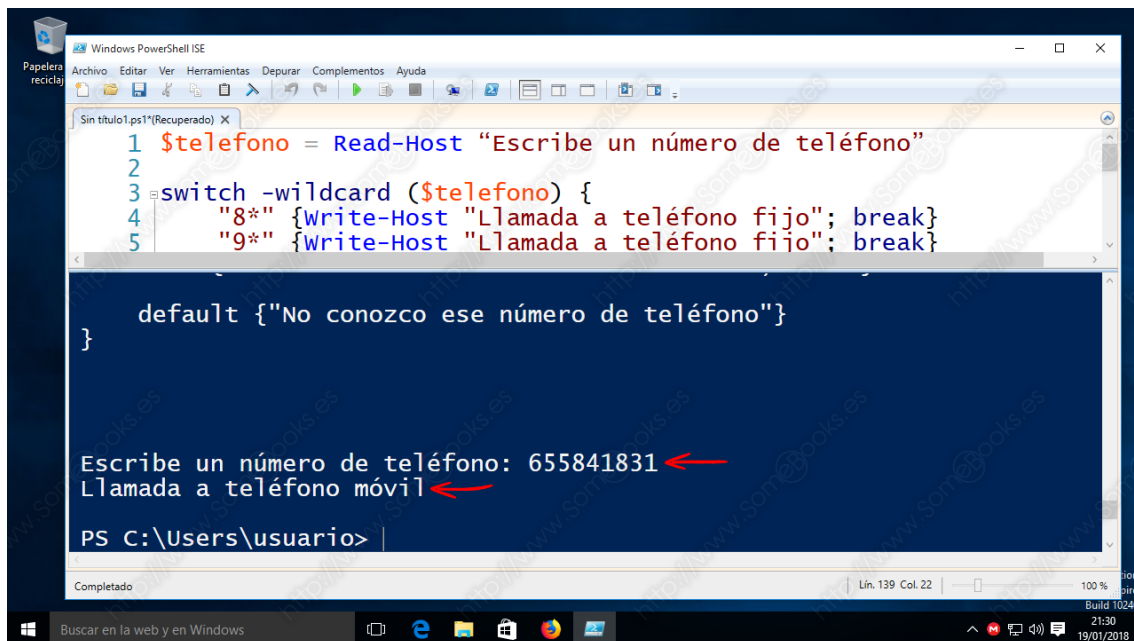
Veamos un ejemplo:

```
$telefono = Read-Host "Escribe un número de teléfono"

switch -wildcard ($telefono) {
    "8*" {Write-Host "Llamada a teléfono fijo"; break}
    "9*" {Write-Host "Llamada a teléfono fijo"; break}
    "6*" {Write-Host "Llamada a teléfono móvil"; break}

    default {"No conozco ese número de teléfono"}
}
```

Al probarlo, obtendríamos algo así:



```
Sin título1.ps1 (Recuperado) X
1 $telefono = Read-Host "Escribe un número de teléfono"
2
3 switch -wildcard ($telefono) {
4     "8*" {Write-Host "Llamada a teléfono fijo"; break}
5     "9*" {Write-Host "Llamada a teléfono fijo"; break}

    default {"No conozco ese número de teléfono"}
}

Escribe un número de teléfono: 655841831
Llamada a teléfono móvil

PS C:\Users\usuario>
```

Punto 6: Estructuras repetitivas

Introducción

Todos los lenguajes de programación necesitan un método que les permita repetir un bloque de instrucciones. En ese sentido, *PowerShell* dispone de una gran variedad de estructuras que le permiten completar esa necesidad. Son las siguientes:

- **do while:** Repite un bloque de código mientras la condición que lo controla siga devolviendo el valor \$true. La condición se evalúa al final del bloque.
- **while:** Repite un bloque de código mientras la condición que lo controla siga devolviendo el valor \$true. La condición se evalúa al principio del bloque.
- **do until:** Repite un bloque de código hasta que la condición que lo controla devuelva el valor \$true. La condición se evalúa al final del bloque.
- **for:** Repite un bloque de código durante un número determinado de veces.
- **foreach:** Repite un bloque de código una vez para cada elemento de una lista

La estructura do while

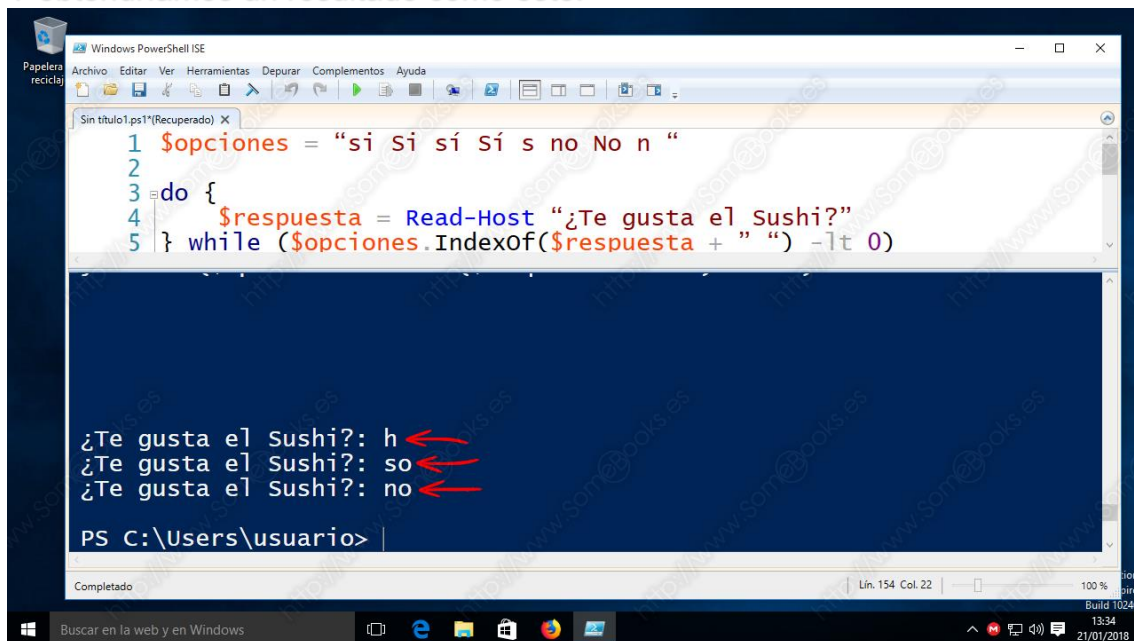
Como hemos dicho más arriba, la estructura do while nos permite repetir un bloque de código mientras una determinada condición siga ofreciendo el valor \$true. Su formato es como sigue:

```
do {Bloque de código}
while (condición)
```

Por ejemplo, cuando pidamos una respuesta al usuario, podemos usar una estructura de este tipo para obligarlo a que dicha respuesta se encuentre dentro de las opciones que nosotros esperamos. Podríamos hacerlo así:

```
$opciones = "sí Si sí Sí s no No n "
do {
    $respuesta = Read-Host "¿Te gusta el sushi?"
} while ($opciones.IndexOf($respuesta + " ") -lt 0)
```

Y obtendríamos un resultado como este:



```
Sin título1.ps1*(Recuperado) X
1 $opciones = "sí Si sí Sí s no No n "
2
3 do {
4     $respuesta = Read-Host "¿Te gusta el sushi?"
5 } while ($opciones.IndexOf($respuesta + " ") -lt 0)

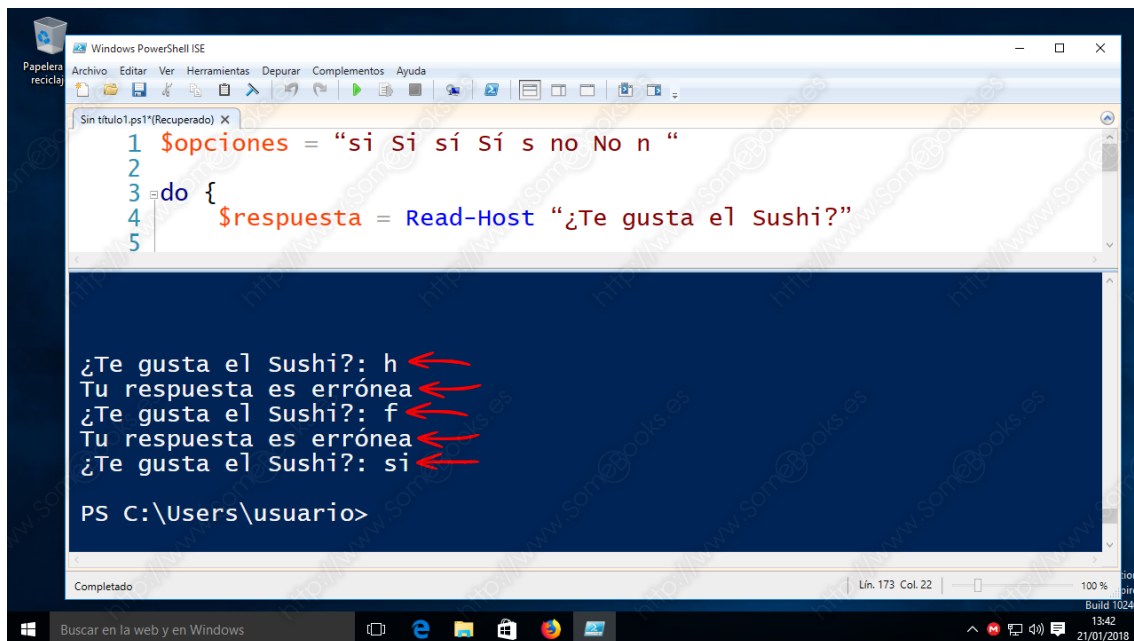
¿Te gusta el sushi?: h
¿Te gusta el sushi?: so
¿Te gusta el sushi?: no

PS C:\Users\usuario>
```

Si quisiéramos explicarle al usuario el motivo por el que volvemos a hacerle la respuesta, deberíamos añadir algo más de código:

```
$opciones = "sí Si sí Sí s no No n "
do {
    $respuesta = Read-Host "¿Te gusta el sushi?"
    if ($opciones.IndexOf($respuesta + " ") -lt 0) {
        Write-Host "Tu respuesta es errónea"
    }
} while ($opciones.IndexOf($respuesta + " ") -lt 0)
```

Y el resultado sería así:



```
Sin título1.ps1 (Recuperado) X
1 $opciones = "si Si sí Sí s no No n "
2
3 do {
4     $respuesta = Read-Host "¿Te gusta el Sushi?"
5 } while ($respuesta -notin $opciones)

¿Te gusta el Sushi?: h
Tu respuesta es errónea
¿Te gusta el Sushi?: f
Tu respuesta es errónea
¿Te gusta el Sushi?: si

PS C:\Users\usuario>
```

Una característica diferenciadora de la estructura *do while* es que se ejecutará el bloque de código que contiene al menos una vez.

La estructura while

Como en el caso anterior, la estructura *while* nos permite repetir un bloque de código mientras la condición que lo controla siga devolviendo el valor *\$true*. Su formato es como sigue:

```
while (condición) {Bloque de código}
```

A diferencia de la estructura anterior, en este caso la condición se evalúa antes del bloque. Esto significa que, si la primera vez la condición ofrece el valor *\$false*, el bloque no se llegará a ejecutar.

Como ejemplo, veamos cómo resolver la situación anterior utilizando la estructura repetitiva *while*:

```
$opciones = "si Si sí Sí s no No n "
$respuesta = Read-Host "¿Te gusta el Sushi?"

while ($opciones.IndexOf($respuesta + " ") -lt 0){
    Write-Host "Tu respuesta es errónea"
    $respuesta = Read-Host "¿Te gusta el Sushi?"
}
```

Y, como vemos, el resultado es equivalente:


```
1 $opciones = "sí Si sí Sí s no No n "  
2  
3 $respuesta = Read-Host "¿Te gusta el Sushi?"  
4  
5 while ($opciones.IndexOf($respuesta + " ") -lt 0){  
  
¿Te gusta el Sushi?: j  
Tu respuesta es errónea  
¿Te gusta el Sushi?: y  
Tu respuesta es errónea  
¿Te gusta el Sushi?: r  
Tu respuesta es errónea  
¿Te gusta el Sushi?: s  
  
PS C:\Users\usuario>
```

La estructura do until

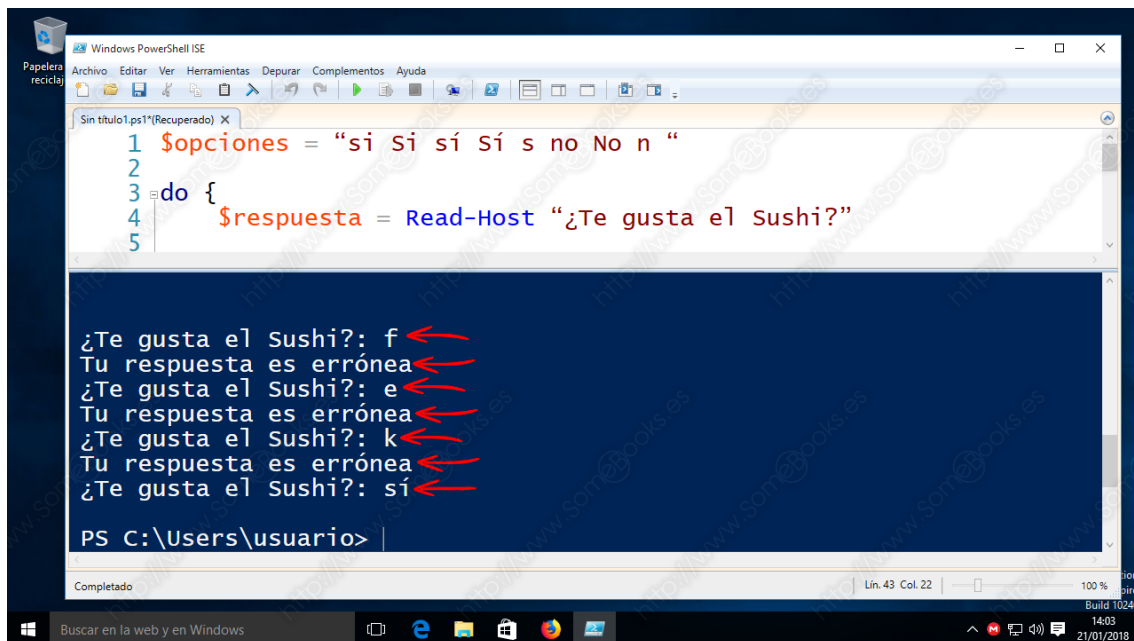
De nuevo, se trata de una variante con respecto a *do while*. La única diferencia es que, en este caso, el bloque de código se repetirá tantas veces como sea necesario para que una determinada condición ofrezca el valor \$true. Su formato es como sigue:

```
do {Bloque de código}  
until (condición)
```

En realidad, su funcionamiento es idéntico a *do while*, pero cambiando el sentido de la condición:

```
$opciones = "sí Si sí Sí s no No n "  
  
do {  
    $respuesta = Read-Host "¿Te gusta el Sushi?"  
    if ($opciones.IndexOf($respuesta + " ") -lt 0) {  
        Write-Host "Tu respuesta es errónea"  
    }  
} until ($opciones.IndexOf($respuesta + " ") -ge 0)
```

Si lo comprobamos, obtendremos algo así:



```
Sin título1.ps1(Recuperado) X
1 $opciones = "sí Si sí Sí s no No n "
2
3 do {
4     $respuesta = Read-Host "¿Te gusta el Sushi?"
5 } while ($?)

¿Te gusta el Sushi?: f
Tu respuesta es errónea
¿Te gusta el Sushi?: e
Tu respuesta es errónea
¿Te gusta el Sushi?: k
Tu respuesta es errónea
¿Te gusta el Sushi?: sí

PS C:\Users\usuario>
```

En realidad, bastaría con poner un operador *-not* delante de la condición que incluíamos en el *do while* para poder utilizarla en *do until*.

```
$opciones = "sí Si sí Sí s no No n "

do {
    $respuesta = Read-Host "¿Te gusta el Sushi?"
    if ($opciones.IndexOf($respuesta + " ") -lt 0) {
        Write-Host "Tu respuesta es errónea"
    }
} until (-not($opciones.IndexOf($respuesta + " ") -lt 0))
```

La estructura for

En las estructuras anteriores, el programador desconoce a priori el número de veces que va a repetirse un determinado bloque. En *do while* y *do until*, lo máximo que podemos decir es que se ejecutarán al menos una vez.

Por el contrario, la orden *for* está pensada para que definamos, desde el principio, el número de repeticiones que van a llevarse a cabo. No obstante, se trata de una estructura muy flexible, que nos permitirá ajustar su comportamiento a nuestras necesidades. Incluso podremos conseguir que se comporte como una estructura *while*.

Su sintaxis general es la siguiente:

```
for (inicialización; condición; incremento)
{Bloque de código}
```

Para entenderla, veamos lo que significa cada uno de los argumentos que aparecen entre paréntesis:

- **inicialización**: serán una o más instrucciones, separadas por comas, que se ejecutarán sólo una vez antes de comenzar la repetición. Normalmente se utiliza para inicializar una variable que se utilizará en el control de la repetición.
- **condición**: se evaluará antes de cada repetición (incluida la primera) y sólo se ejecutará el bloque cuando su valor sea `$true`. Por lo tanto, la repetición terminará la primera vez que la condición devuelva el valor `$false`.
- **incremento**: serán una o más instrucciones, separadas por comas, que se ejecutarán al final de cada repetición. Esta parte suele utilizarse para modificar la variable que será comprobada en la **condición** para controlar las repeticiones del bucle.

Probablemente, la mejor forma de entenderlo sea a través de un ejemplo. Veamos cómo mostrar en pantalla los números del 1 al 5:

```
for ($i=1; $i -le 5; $i++){
    Write-Host $i
}
```

Su funcionamiento será el siguiente:

- Comienza ejecutando la **inicialización**, que se hará sólo la primera vez (almacena el valor 1 en la variable `$i`).
- A continuación, se evalúa la **condición** (comprueba si el contenido de la variable `$i` es inferior o igual a 5).
- Si la **condición** devuelve `$true`, se ejecuta el **bloque de código** (en este caso el **cmdlet** `Write-Host`) y se produce el **incremento** (se le suma 1 al valor de `$i`).
- Después, volverá a evaluarse la **condición**, repitiendo todo el proceso.

La repetición continuará hasta que la condición devuelva el valor `$false`. En ese caso, la ejecución continuará con la instrucción que haya después de la estructura **for**.

Debes tener en cuenta que los argumentos de **for** van separados por caracteres de punto y coma (;)

Como puedes suponer, el resultado del código anterior será como el que se muestra en la siguiente imagen:

La estructura **foreach**

Es una estructura bastante especial, diseñada para recorrer todo tipo de listas de elementos. Normalmente, mientras se ejecuta el **bloque de código**, se aplica alguna operación sobre cada elemento de la lista. Su formato general será así:

```
foreach (elemento in colección)
{Bloque de código}
```

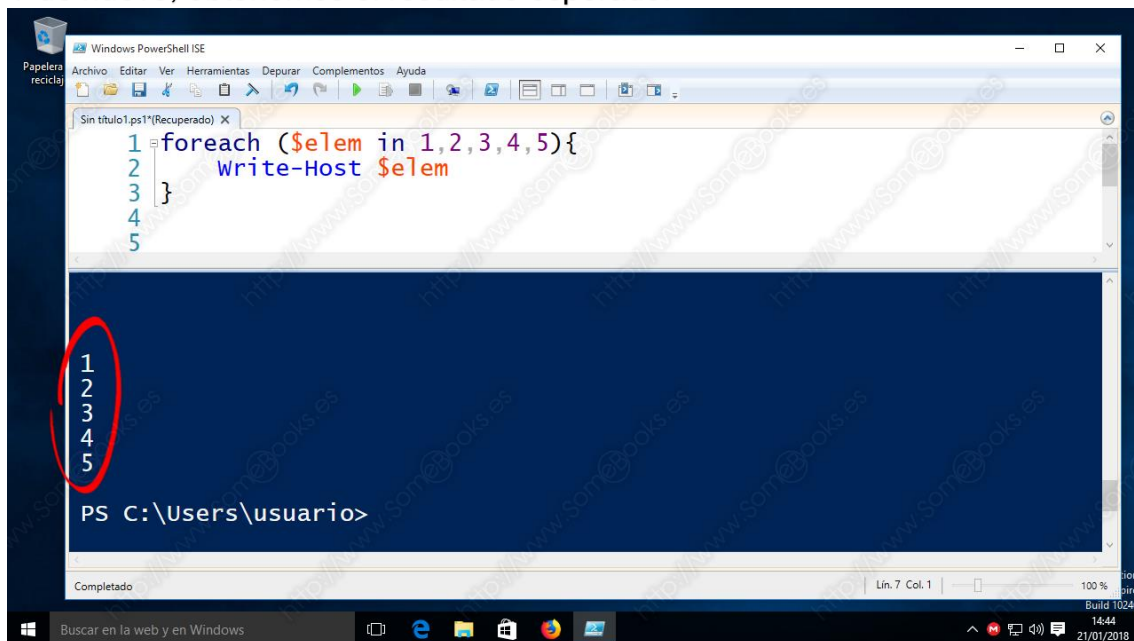
Para entenderla, veamos lo que significa cada uno de los argumentos que aparecen entre paréntesis:

- *elemento*: representa una variable que tomará, en cada repetición y de forma ordenada, el valor de uno de los elementos de la *colección*.
- *colección*: será el conjunto de valores sobre los que actuará el *bloque de código*.

Probablemente, la mejor forma de entenderlo sea a través de un ejemplo. Veamos cómo mostrar en pantalla los números del 1 al 5:

```
foreach ($elem in 1,2,3,4,5){
    Write-Host $elem
}
```

Y de nuevo, obtenemos el resultado esperado:



```
Sin título1.ps1*(Recuperado) X
1 foreach ($elem in 1,2,3,4,5){
2   Write-Host $elem
3 }
4
5

1
2
3
4
5

PS C:\Users\usuario>

Completado
```

Cuando la lista es muy larga, podrías utilizar también intervalos:


```
foreach ($elem in 1..5){
    Write-Host $elem
}
```

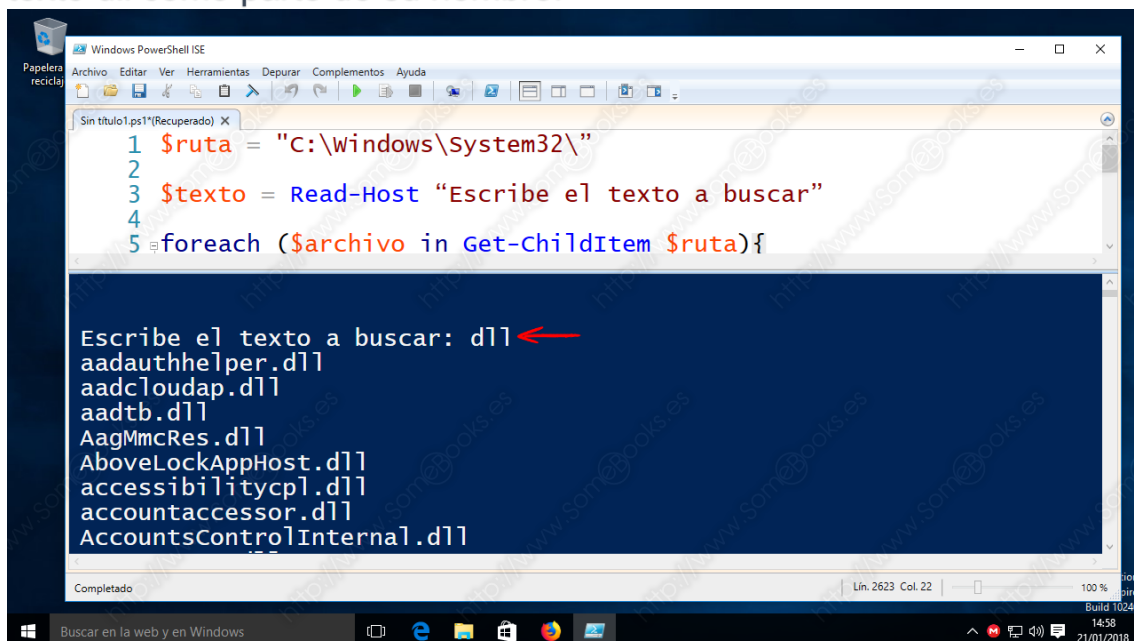
En realidad, *foreach* funciona en cualquier situación donde pueda obtenerse una lista de elementos. Por ejemplo, podríamos utilizar el *cmdlet* *Get-ChildItem*, que permite obtener la lista de archivos de una o varias rutas del disco, para averiguar qué nombres de archivo, de una determinada carpeta, contienen el texto que hayamos escrito:

```
$ruta = "C:\windows\System32\"

$texto = Read-Host "Escribe el texto a buscar"

foreach ($archivo in Get-ChildItem $ruta){
    if ($archivo.Name.IndexOf($texto) -ge 0) {
        Write-Host $archivo.Name
    }
}
```

Por ejemplo, podemos obtener todos los archivos que contengan el texto *dll* como parte de su nombre:

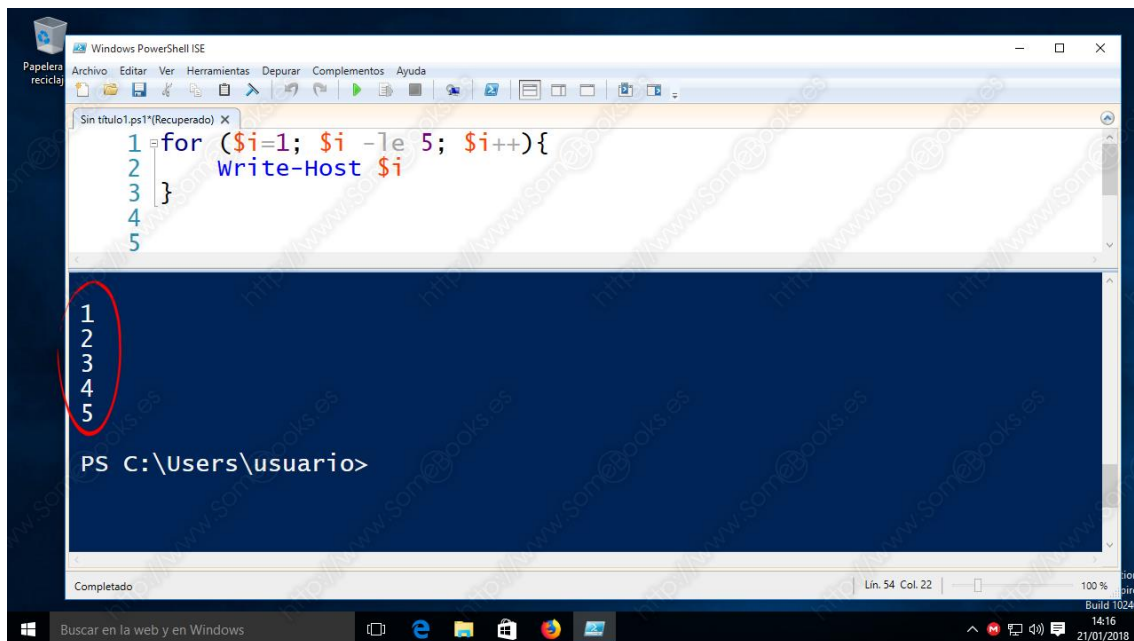


```
Sin título1.ps1*(Recuperado) X
1 $ruta = "C:\windows\System32\"
2
3 $texto = Read-Host "Escribe el texto a buscar"
4
5 foreach ($archivo in Get-ChildItem $ruta){

Escribe el texto a buscar: dll
aadauthhelper.dll
aadcloudap.dll
aadtb.dll
AagMmcRes.dll
AboveLockAppHost.dll
accessibilitycp1.dll
accountaccessor.dll
AccountsControlInternal.dll

Completado Lin. 2623 Col. 22 100 % Build 10240 14:58 21/01/2018
```

... Aunque en la imagen sólo vemos el principio de la lista, que es bastante larga



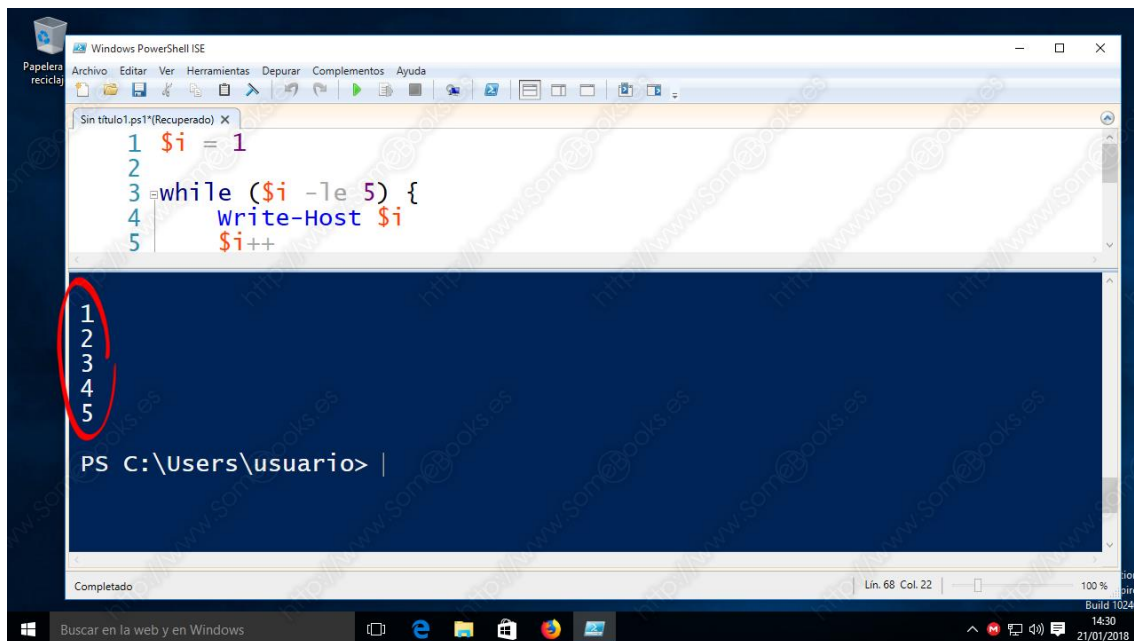
The screenshot shows the Windows PowerShell ISE interface. The script editor at the top contains a `for` loop: `1 for ($i=1; $i -le 5; $i++){`, `2 Write-Host $i`, `3 }`, `4`, `5`. The console window below shows the output of the script: `1`, `2`, `3`, `4`, `5`. The numbers 1 through 5 are listed vertically, with the first four numbers (1, 2, 3, 4) circled in red. The console prompt is `PS C:\Users\usuario>`. The status bar at the bottom indicates 'Completado' and 'Lin. 54 Col. 22'.

En realidad, es muy sencillo escribir un bucle *while* que se comporte como un bucle *for*. Por ejemplo, el ejemplo anterior podríamos escribirlo de la siguiente forma:

```
$i = 1

while ($i -le 5) {
    Write-Host $i
    $i++
}
```

Y el funcionamiento seguirá siendo idéntico:



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

```
1 $i = 1
2
3 while ($i -le 5) {
4     Write-Host $i
5     $i++
6 }
```

The output console shows the numbers 1 through 5, each on a new line, indicating the loop executed five times. A red circle highlights the output numbers. The status bar at the bottom indicates 'Completado' (Completed) and 'Lin. 68 Col. 22'.

Como puedes ver, ha bastado con poner la *inicialización* justo encima del bucle y el *incremento* como última instrucción del *bloque de código*. Incluso podemos hacer que un bucle *for* se comporte como un bucle *while*. Basta con dejar vacíos la *inicialización* y el *incremento*:

```
$i = 1

for (;$i -le 5;) {
    Write-Host $i
    $i++
}
```

Alterar el proceso de las estructuras repetitivas

En *PowerShell* tenemos dos órdenes que nos permiten modificar la ejecución normal de una estructura repetitiva: *break* y *continue*.

Veamos cómo funcionan.

La orden *break*

Al ejecutarla, *PowerShell* deja de ejecutar el resto de instrucciones del *bloque de código* y termina la repetición del bucle aunque no se haya cumplido la condición de salida. La ejecución continuará por la siguiente instrucción después del *bloque de código* que se está repitiendo (si no la hay, acabará el *script*).

Veamos un ejemplo que mostrará los números del 1 al 100, salvo que el usuario decida interrumpirlo:

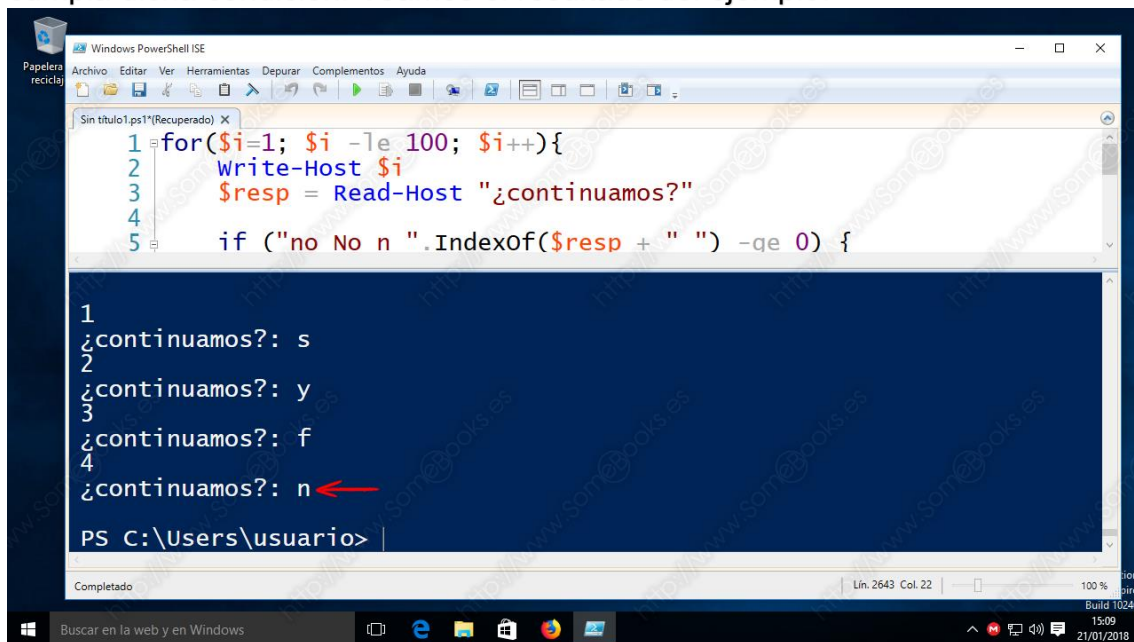
```

for($i=1; $i -le 100; $i++){
    Write-Host $i
    $resp = Read-Host "¿continuamos?"

    if ("no No n ".IndexOf($resp + " ") -ge 0) {
        break
    }
}

```

Como en el ejemplo, lo normal es que la orden *break* se encuentre dentro de una condición. De este modo, el bucle se detendrá en el momento que se cumpla dicha condición. Veamos el resultado del ejemplo:



```

1 for($i=1; $i -le 100; $i++){
2     Write-Host $i
3     $resp = Read-Host "¿continuamos?"
4
5     if ("no No n ".IndexOf($resp + " ") -ge 0) {
6
7     }
8 }

```

1 ¿continuamos?: s
2 ¿continuamos?: y
3 ¿continuamos?: f
4 ¿continuamos?: n
PS C:\Users\usuario>

Esta orden puede aplicarse tanto en las estructuras repetitivas (*do while*, *while*, *do until*, *for* y *foreach*) como a la estructura alternativa *switch*. En el caso de *switch*, al ejecutar la orden *break* el flujo de ejecución también continuará por la instrucción que haya después de *switch* (y si no la hay, acabará el *script*).

La orden *continue*

Como ocurre con *break*, al ejecutar la orden *continue* se interrumpe la ejecución del bloque de código que la contiene. La diferencia es que, en este caso, la ejecución sigue con la siguiente repetición del bucle. Por lo tanto, estamos consiguiendo saltarnos las instrucciones que quedaban por ejecutar en el bloque de código.

Por ejemplo, supongamos que necesitamos pedir 10 números al usuario, pero que sólo se muestren los que sean pares:

```

for($i=1; $i -le 10; $i++){
    [int]$num = Read-Host "Escribe un número"

    if ($num %2 -eq 1) { continue }

    Write-Host "El número $num es par"
}

```

Y el resultado obtenido podría ser como este:

```

1 for($i=1; $i -le 10; $i++){
2     [int]$num = Read-Host "Escribe un número"
3
4     if ($num %2 -eq 1) { continue }
5
Escribe un número: 7
Escribe un número: 5
Escribe un número: 3
Escribe un número: 9
Escribe un número: 8
El número 8 es par
Escribe un número: 1
Escribe un número: 13
Escribe un número: 2
El número 2 es par
Escribe un número:

```

Punto 7: Variables que guardan múltiples valores

Introducción

En *PowerShell* tenemos dos formas diferentes de crear variables que contengan colecciones de datos: los *Arrays* y las *tablas Hash*. Estas últimas también suelen recibir el nombre de *arrays asociativos*. En este apartado hablaremos de ambas.

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 = @(valor 1, valor 2, valor 3)
```

Por ejemplo, el siguiente código sería correcto:

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

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

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

o, incluso, usando un rango de valores:

```
$a = 1..4
```

Como hemos dicho más arriba, 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 hemos utilizado el formato estándar, la definición explícita de tipo es válida también con la sintaxis reducida que hemos visto antes:

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

o también:

```
[int[]] $a = 1..4
```

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

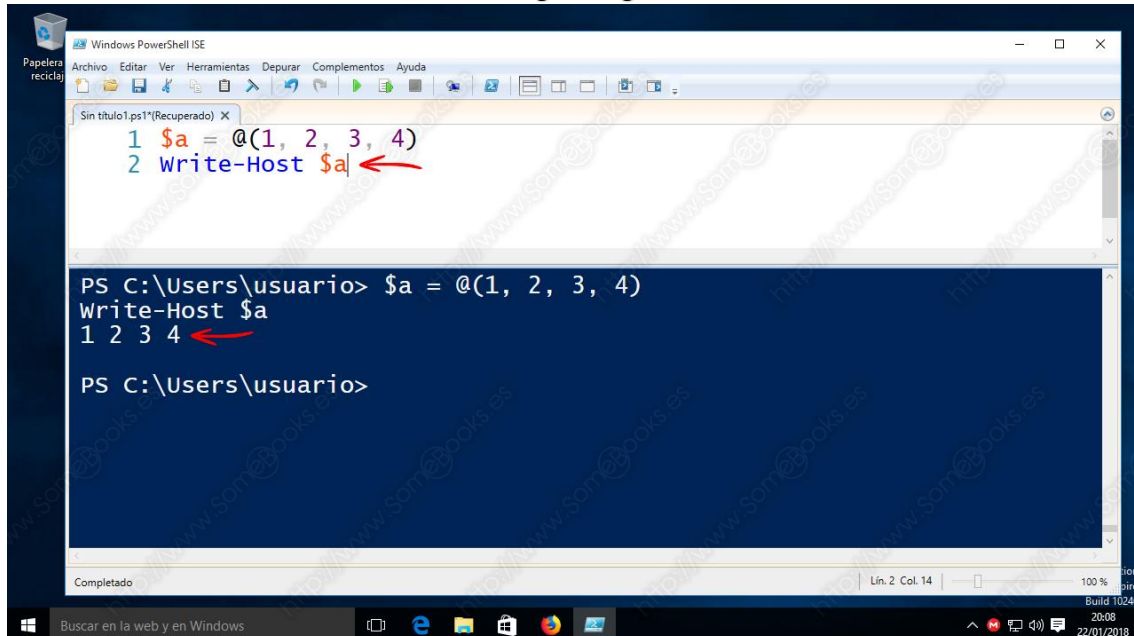
```
$a = @()
```

[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
```

Y el resultado será como el de la imagen siguiente.



```
Sin título1.ps1*(Recuperado) X
1 $a = @(1, 2, 3, 4)
2 Write-Host $a

PS C:\Users\usuario> $a = @(1, 2, 3, 4)
Write-Host $a
1 2 3 4

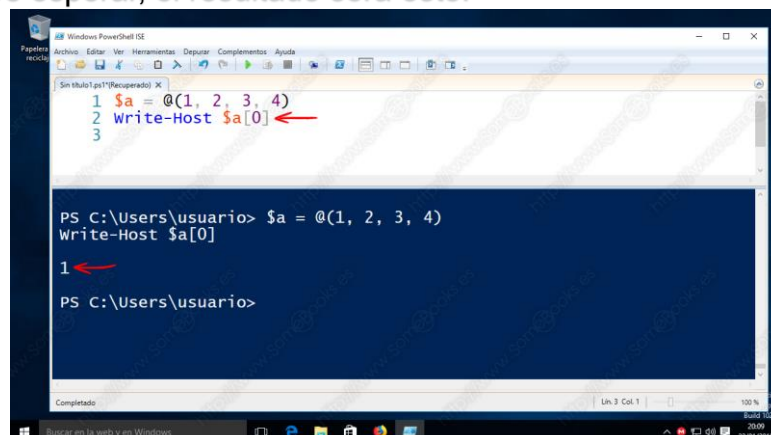
PS C:\Users\usuario>
```

Utilizar 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]
```

Y, como cabe esperar, el resultado será este:



```
Sin título1.ps1*(Recuperado) X
1 $a = @(1, 2, 3, 4)
2 Write-Host $a[0]
3

PS C:\Users\usuario> $a = @(1, 2, 3, 4)
Write-Host $a[0]
1

PS C:\Users\usuario>
```

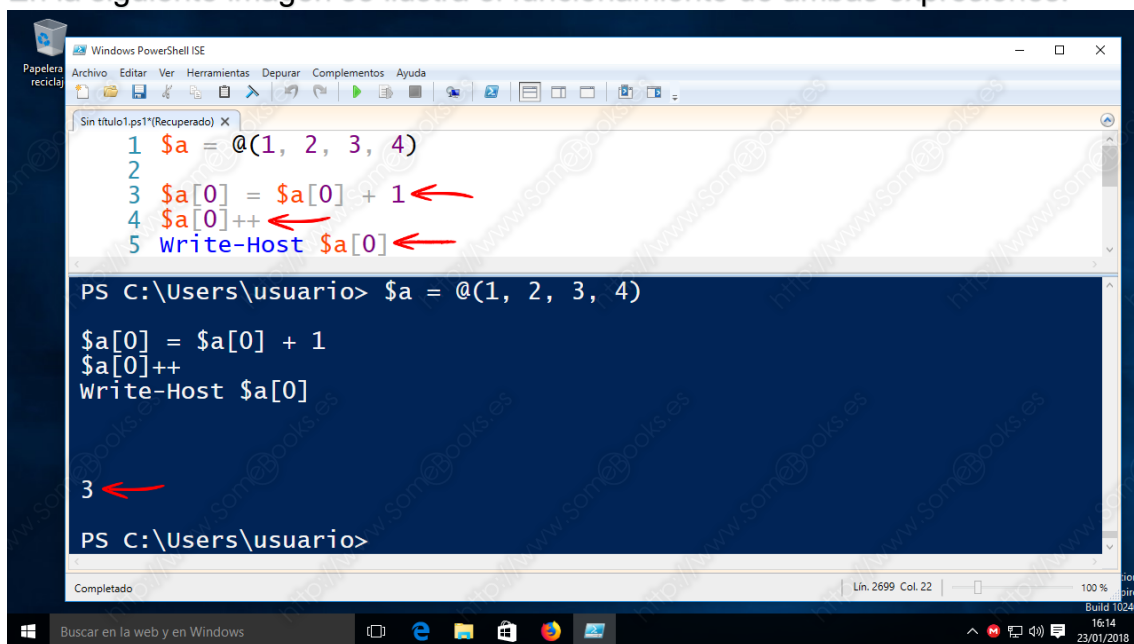
Aunque, si lo que queremos es utilizar el elemento dentro de una expresión, nos limitamos a utilizar la notación anterior como si se tratara de una variable simple. Por ejemplo, para incrementar en uno el valor del elemento, haríamos esto:

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

o incluso esto:

```
$a[0]++
```

En la siguiente imagen se ilustra el funcionamiento de ambas expresiones:



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

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

Red arrows point from the code lines to the output console. The console shows the execution of the script:

```
PS C:\Users\usuario> $a = @(1, 2, 3, 4)

$a[0] = $a[0] + 1
$a[0]++
Write-Host $a[0]

3
```

The output is the number 3, which is the value of the first element of the array after the two increment operations.

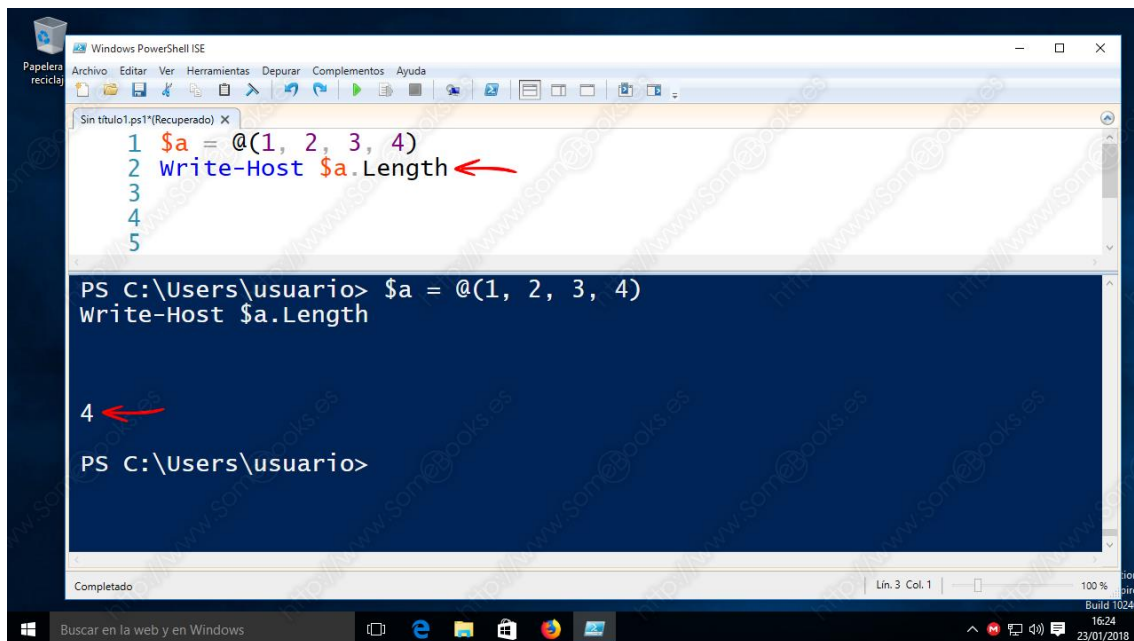
Arrays como objetos

Ya dijimos en otro apartado que, internamente, *PowerShell* trata a todas las variables como objetos. Y, aunque aquí no profundicemos en los conceptos relacionados con la programación orientada a objetos, sí debemos saber que, gracias a esta característica, disponemos de herramientas que nos facilitan la creación, manipulación, búsqueda y ordenación de *arrays*.

Por ejemplo, si quisiéramos saber cuántos elementos tiene un *array*, bastaría con escribir lo siguiente:

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

Y, como vemos a continuación, el valor devuelto para el array de ejemplo que hemos venido utilizando, sería 4:

A screenshot of the Windows PowerShell ISE interface. The editor window shows a script with two lines: `$a = @(1, 2, 3, 4)` and `Write-Host $a.Length`. A red arrow points to the `Length` property. The console window below shows the execution of these commands, resulting in the output `4`. Another red arrow points to the output `4`. The status bar at the bottom indicates 'Completado' and 'Lin. 3 Col. 1'.

Para conocer en detalle todas las características de la clase *Array*, puedes recurrir al siguiente enlace: [https://msdn.microsoft.com/en-us/library/system.array\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.array(v=vs.110).aspx)

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*. Para entenderlo, quizás lo mejor sea recurrir a un ejemplo; veamos 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
}
```

Observa que 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.)

En la siguiente imagen puedes comprobar el resultado de ejecutar el ejemplo anterior:

```
Sin título1.ps1 (Recuperado) X
1 $enteros = @( 1, 2, 3, 4, 5, 6, 7, 8, 9)
2
3 for ($i = 0; $i -lt $enteros.Length; $i++) {
4     $enteros[$i] = $enteros[$i] * 2
5 }

PS C:\Users\usuario> $enteros = @( 1, 2, 3, 4, 5, 6, 7, 8, 9)

for ($i = 0; $i -lt $enteros.Length; $i++) {
    $enteros[$i] = $enteros[$i] * 2
}
Write-Host $enteros

2 4 6 8 10 12 14 16 18

PS C:\Users\usuario>
```

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 - 1; $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, recuerda que existe una de ellas que está específicamente diseñada para actuar sobre colecciones de datos. Me refiero a la estructura *foreach*.

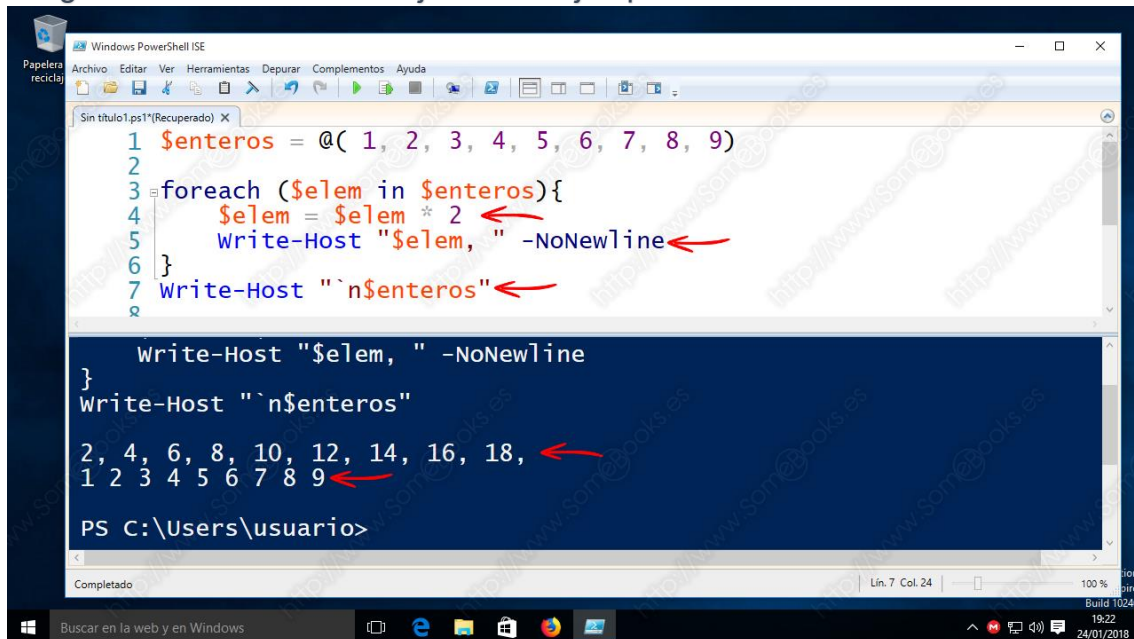
Una de las ventajas de usar *foreach* para recorrer un *array* es que no necesitamos una variable que vaya tomando el valor de cada posición. Por ejemplo, si queremos resolver la tarea anterior, bastaría con el siguiente código:

```
$enteros = @( 1, 2, 3, 4, 5, 6, 7, 8, 9)

foreach ($elem in $enteros){
    $elem = $elem * 2
}
```

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*. Para comprobarlo, observa la siguiente

imagen con el resultado de ejecutar el ejemplo anterior:



```
Sin titulo1.ps1*(Recuperado) X
1 $enteros = @( 1, 2, 3, 4, 5, 6, 7, 8, 9)
2
3 foreach ($elem in $enteros){
4     $elem = $elem * 2
5     Write-Host "$elem, " -NoNewline
6 }
7 Write-Host "`n$enteros"
8
9
10 Write-Host "$elem, " -NoNewline
11 }
12 Write-Host "`n$enteros"
13
14 2, 4, 6, 8, 10, 12, 14, 16, 18,
15 1 2 3 4 5 6 7 8 9
16
17 PS C:\Users\usuario>
```

Observa que, por primera vez, 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* aparezcan en la misma línea. También hemos utilizado un carácter especial (*`n*) en el último *Write-Host*. En este caso, la intención era, 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 = @( "Fantasma, en un tarro", [float] 120,
                "Detector de ovnis", [float] 150,
                "Significado de la vida", [float] 30,
                "Chicle de Britney Spears, usado", [float] 14000)

for ($i = 1; $i -lt $productos.Length; $i+=2) {
    $productos[$i] = $productos[$i] * 0.9
}
```

Como *PowerShell* no tiene en cuenta los saltos de línea ni el exceso de espacios, hemos hecho nuestro 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:

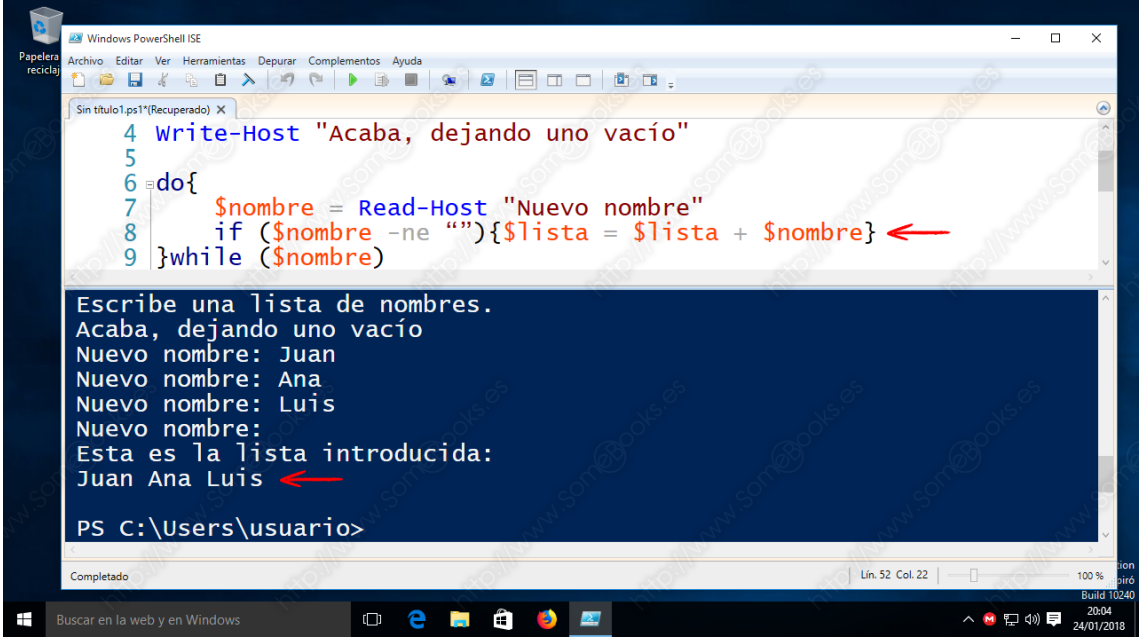
```
$lista = @()

Write-Host "Escribe una lista de nombres."
Write-Host "Acaba, dejando uno vacío"

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

Write-Host "Esta es la lista introducida:"
Write-Host $lista
```

Su ejecución podría ser como sigue:

A screenshot of the Windows PowerShell ISE interface. The script editor shows a script with line numbers 4 to 9. Line 4: Write-Host "Acaba, dejando uno vacío". Line 5: Empty line. Line 6: do{. Line 7: \$nombre = Read-Host "Nuevo nombre". Line 8: if (\$nombre -ne ""){\$lista = \$lista + \$nombre}. Line 9: }while (\$nombre). A red arrow points to the '+' operator in line 8. The console window below shows the output: "Escribe una lista de nombres.", "Acaba, dejando uno vacío", "Nuevo nombre: Juan", "Nuevo nombre: Ana", "Nuevo nombre: Luis", "Nuevo nombre:", "Esta es la lista introducida:", and "Juan Ana Luis". A red arrow points to the output "Juan Ana Luis". The status bar at the bottom indicates "Completado", "Lin. 52 Col. 22", "100%", and "Build 10240". The taskbar at the very bottom shows the Windows logo, a search bar, and several application icons.

```
Sin título1.ps1*(Recuperado) X
4 Write-Host "Acaba, dejando uno vacío"
5
6 do{
7     $nombre = Read-Host "Nuevo nombre"
8     if ($nombre -ne ""){$lista = $lista + $nombre}
9 }while ($nombre)

Escribe una lista de nombres.
Acaba, dejando uno vacío
Nuevo nombre: Juan
Nuevo nombre: Ana
Nuevo nombre: Luis
Nuevo nombre:
Esta es la lista introducida:
Juan Ana Luis

PS C:\Users\usuario>
```

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

```
$a = @(1, 2, 3, 4, 5)
$b = @(6, 7, 8, 9, 10)

$c = $a + $b
```

Cuando comprobamos el resultado obtenemos algo así:

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

```
1 $a = @(1, 2, 3, 4, 5)
2 $b = @(6, 7, 8, 9, 10)
3
4 $c = $a + $b
5
6 Write-Host "Este es el resultado:"
7 Write-Host $c
```

The output console shows the result of the execution:

```
Este es el resultado:
1 2 3 4 5 6 7 8 9 10
PS C:\Users\usuario>
```

Red arrows point to the `$c = $a + $b` line in the script and the output `1 2 3 4 5 6 7 8 9 10` in the console.

... Incluso podemos añadir una lista a otra:

```
$a = @(1, 2, 3, 4, 5)
$b = @(6, 7, 8, 9, 10)

$a += $b
```

Y comprobamos que también funciona correctamente:

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

```
1 $a = @(1, 2, 3, 4, 5)
2 $b = @(6, 7, 8, 9, 10)
3
4 $a += $b
5
6 Write-Host "Este es el resultado:"
7 Write-Host $a
```

The output console shows the result of the execution:

```
Este es el resultado:
1 2 3 4 5 6 7 8 9 10
PS C:\Users\usuario>
```

Red arrows point to the `$a += $b` line in the script and the output `1 2 3 4 5 6 7 8 9 10` in the console.

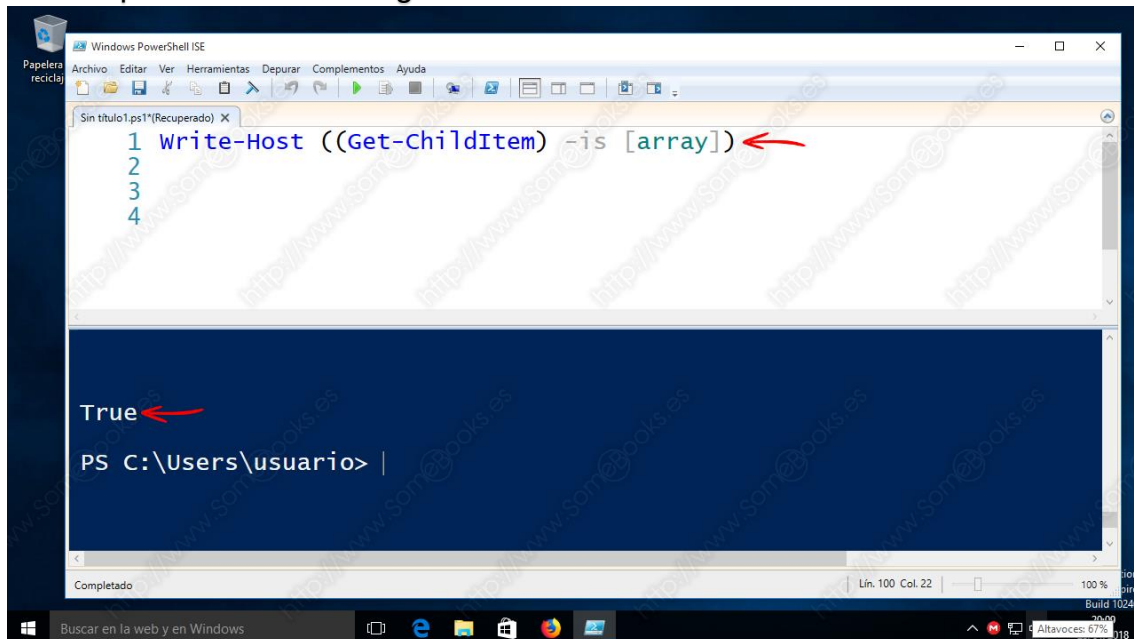
En ocasiones, la salida de un *cmdlet* consiste en un *array*. Esto significa que podríamos asignarla directamente a una variable de este tipo. Un ejemplo puede ser el *cmdlet* *Get-ChildItem* que ya conocemos del capítulo anterior.

Get-ChildItem 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])
```

Con el que obtenemos el siguiente resultado:

A screenshot of the Windows PowerShell ISE interface. The command prompt shows the command `Write-Host ((Get-ChildItem) -is [array])` on line 1, with a red arrow pointing to the `[array]` parameter. The output of the command is `True`, also indicated by a red arrow. The PowerShell prompt is `PS C:\Users\usuario> |`. The status bar at the bottom indicates 'Completado' and 'Lin. 100 Col. 22'.

Como el resultado es satisfactorio, podemos utilizar un *array* para almacenar la información que nos ofrece escribiendo algo así:

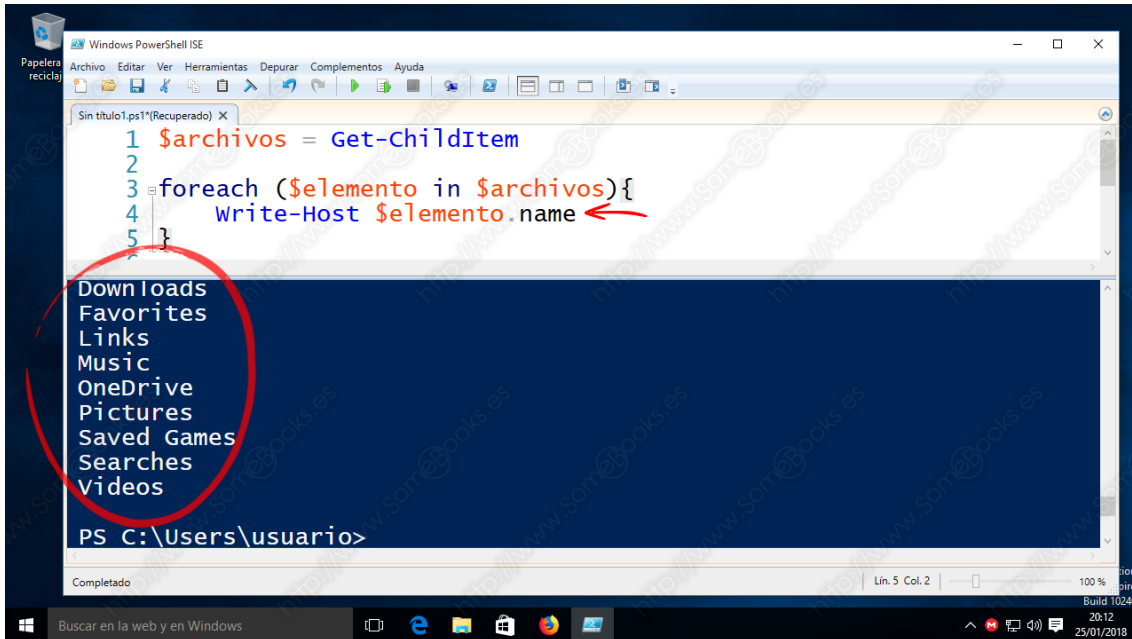
```
$archivos = Get-ChildItem
```

En realidad, *Get-ChildItem* devuelve un *array* de objetos donde cada uno representará un archivo del directorio. A modo de ejemplo, 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. Algo así:

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

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

El resultado será así:



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

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

The output console shows the following results:

```
Downloads
Favorites
Links
Music
OneDrive
Pictures
Saved Games
Searches
Videos
```

The output is highlighted with a red circle. The status bar at the bottom indicates 'Completado' (Completed) and 'Lin. 5 Col. 2'.

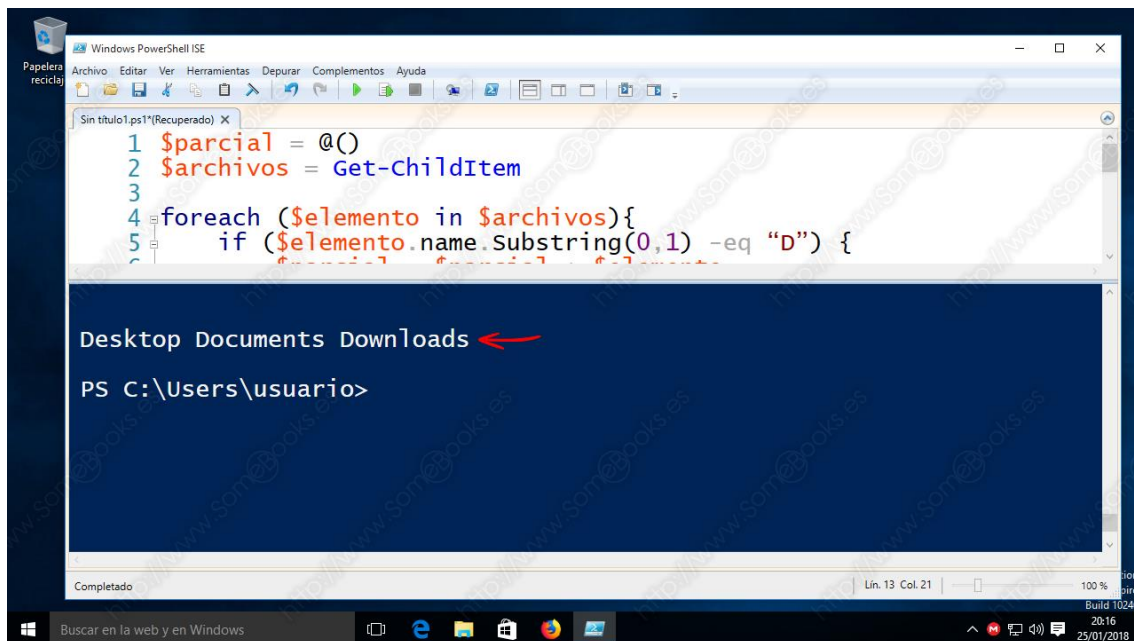
En cuanto a la eliminación de elementos de una *array*, no hay un procedimiento sencillo. Probablemente la mejor solución consista en crear un *array* nuevo y asignarle únicamente los valores que nos interese mantener. Por ejemplo, supongamos que, en el *array* anterior, únicamente queremos mantener 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
```

... Y si comprobamos el resultado, obtendremos algo como esto:



```
Sin título1.ps1*(Recuperado) X
1 $parcial = @()
2 $archivos = Get-ChildItem
3
4 foreach ($elemento in $archivos){
5     if ($elemento.name.Substring(0,1) -eq "D") {
6         $parcial += $elemento
7     }
8 }
9
10 $parcial
```

Desktop Documents Downloads ←

PS C:\Users\usuario>

En realidad, podríamos haber simplificado un poco escribiendo algo así:

```
$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.

De algún modo, podríamos pensar que 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 = @{clave 1 = valor 1; clave 2 = valor 2; clave 3 = valor 3}
```

Veamos el *array* del ejemplo de arriba reinterpretado como tabla *Hash*:

```
$productos = @{
    "Fantasma, en un tarro" = [float] 120;
    "Detector de ovnis" = [float] 150;
    "Significado de la vida" = [float] 30;
    "Chicle de Britney Spears, usado" = [float] 14000
}
```

Igual que hicimos con los *arrays*, aprovechamos que *PowerShell* no tiene en cuenta los saltos de línea ni el exceso espacios, para conseguir que nuestro código más legible añadiéndolos según nuestro criterio.

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


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

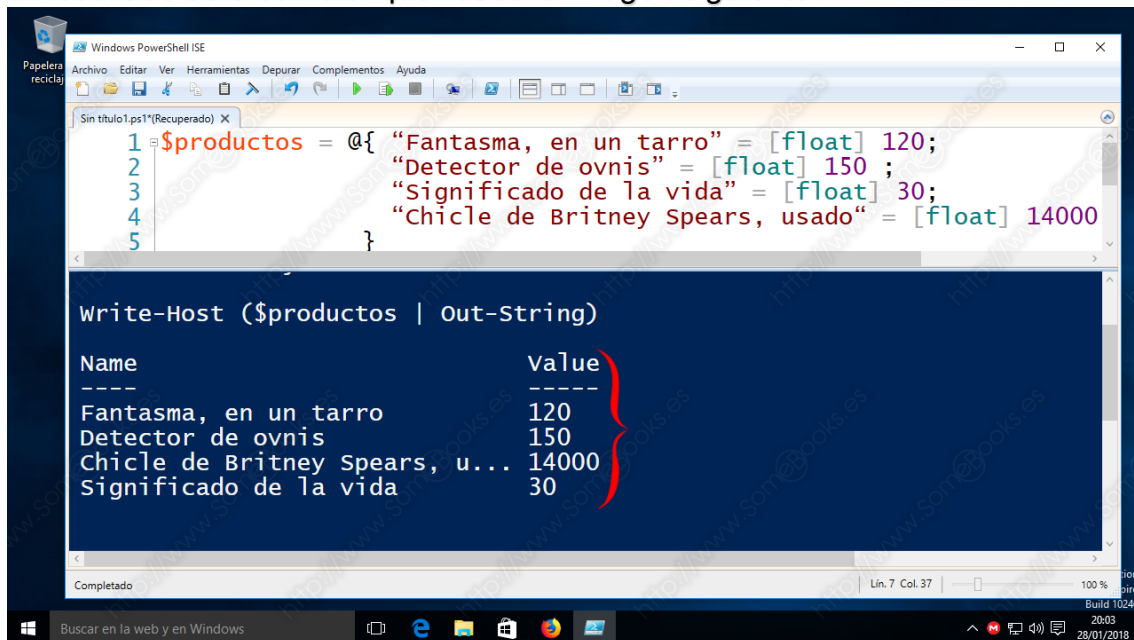
Mostrar el contenido de una tabla Hash

A diferencia de lo que ocurría con los *arrays*, para que *Write-Host* consiga mostrar el contenido de una *tabla Hash*, deberemos 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 será similar a esta:

```
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*:

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

... o sólo los valores, usando la propiedad *Values*:

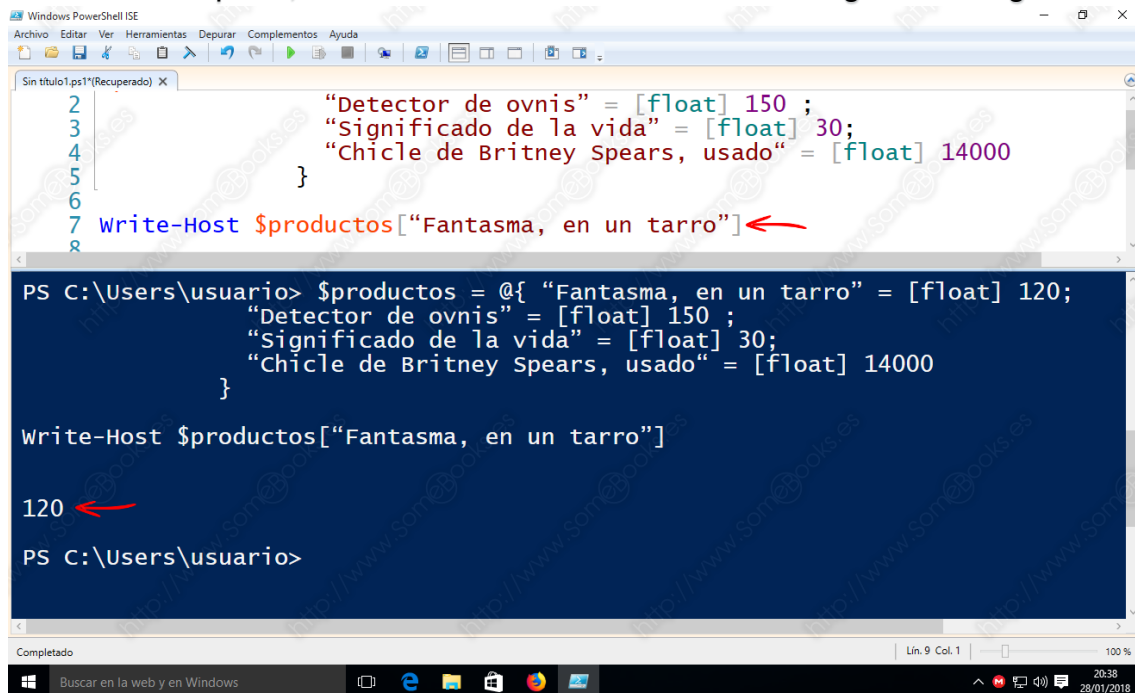
```
Write-Host $productos.Values
```

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["Fantasma, en un tarro"]`

Como cabe esperar, el resultado obtenido es como en la siguiente imagen:



```
Sin título1.ps1 (Recuperado) X
2
3
4
5
6
7 Write-Host $productos["Fantasma, en un tarro"]
8

PS C:\Users\usuario> $productos = @{ "Fantasma, en un tarro" = [float] 120;
    "Detector de ovnis" = [float] 150 ;
    "Significado de la vida" = [float] 30;
    "Chicle de Britney Spears, usado" = [float] 14000
}

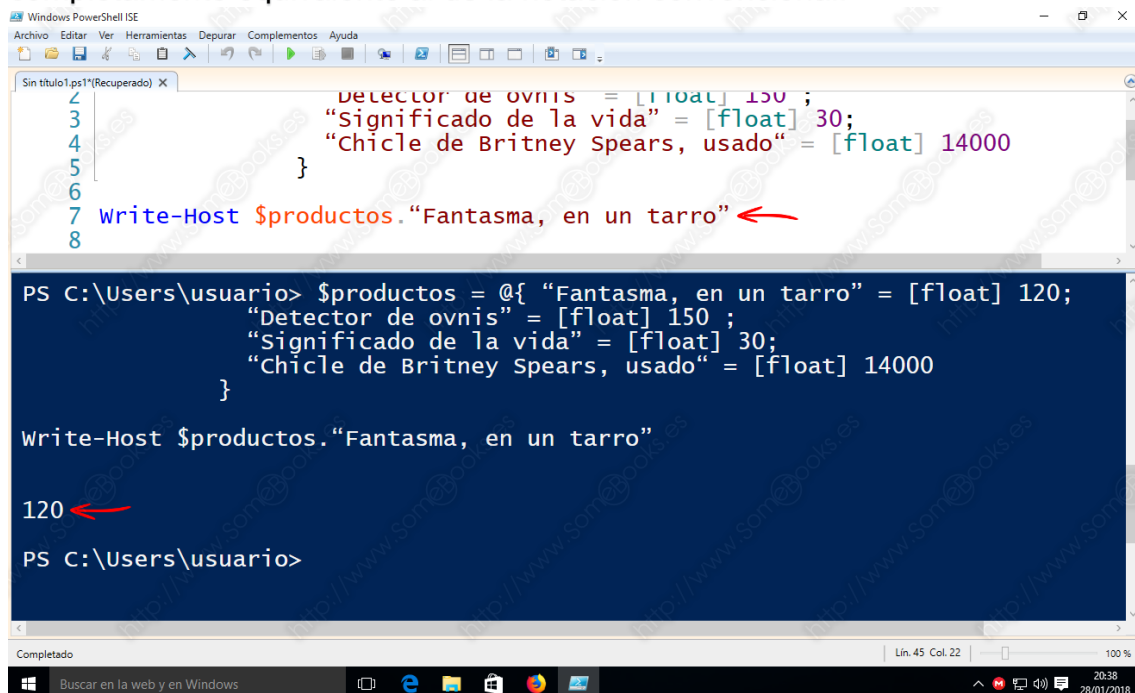
Write-Host $productos["Fantasma, en un tarro"]

120
PS C:\Users\usuario>
```

También podemos utilizar notación orientada a objetos:

`Write-Host $productos."Fantasma, en un tarro"`

Como puedes ver en la imagen siguiente, el resultado que obtenemos es completamente equivalente al de la notación convencional:



```
Sin título1.ps1 (Recuperado) X
2
3
4
5
6
7 Write-Host $productos."Fantasma, en un tarro"
8

PS C:\Users\usuario> $productos = @{ "Fantasma, en un tarro" = [float] 120;
    "Detector de ovnis" = [float] 150 ;
    "Significado de la vida" = [float] 30;
    "Chicle de Britney Spears, usado" = [float] 14000
}

Write-Host $productos."Fantasma, en un tarro"

120
PS C:\Users\usuario>
```

Incluso podemos prescindir de las comillas cuando la clave consiste en una sola palabra. Por ejemplo, si tuviésemos la siguiente *tabla Hash*...

```
$colores = @{"Rojo" = "ff0000";
              "Aceituna" = "808000";
              "Naranja" = "ffa500";
              "Cian" = "00ffff"
            }
```

Write-Host \$colores.Rojo

Lógicamente, el resultado que obtenemos será el valor almacenado en la clave *Rojo*:

```
Sin título1.ps1 (Recuperado) X
1 $colores = @{"Rojo" = "ff0000";
2             "Aceituna" = "808000";
3             "Naranja" = "ffa500";
4             "Cian" = "00ffff"
5         }
6
7 Write-Host $colores.Rojo ←

PS C:\Users\usuario> $colores = @{"Rojo" = "ff0000";
    "Aceituna" = "808000";
    "Naranja" = "ffa500";
    "Cian" = "00ffff"
}
Write-Host $colores.Rojo

ff0000 ←
PS C:\Users\usuario>
```

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, haríamos esto:

```
$productos["Fantasma, en un tarro"] = $productos["Fantasma, en un tarro"] + 1
```

o incluso esto:

```
$productos["Fantasma, en un tarro"]++
```

Tabla Hash como objetos

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 algo así:

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

Para conocer en detalle todas las características de la clase Hashtable, puedes recurrir al siguiente enlace: [https://msdn.microsoft.com/es-es/library/system.collections.hashtable\(v=vs.110\).aspx](https://msdn.microsoft.com/es-es/library/system.collections.hashtable(v=vs.110).aspx).

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.

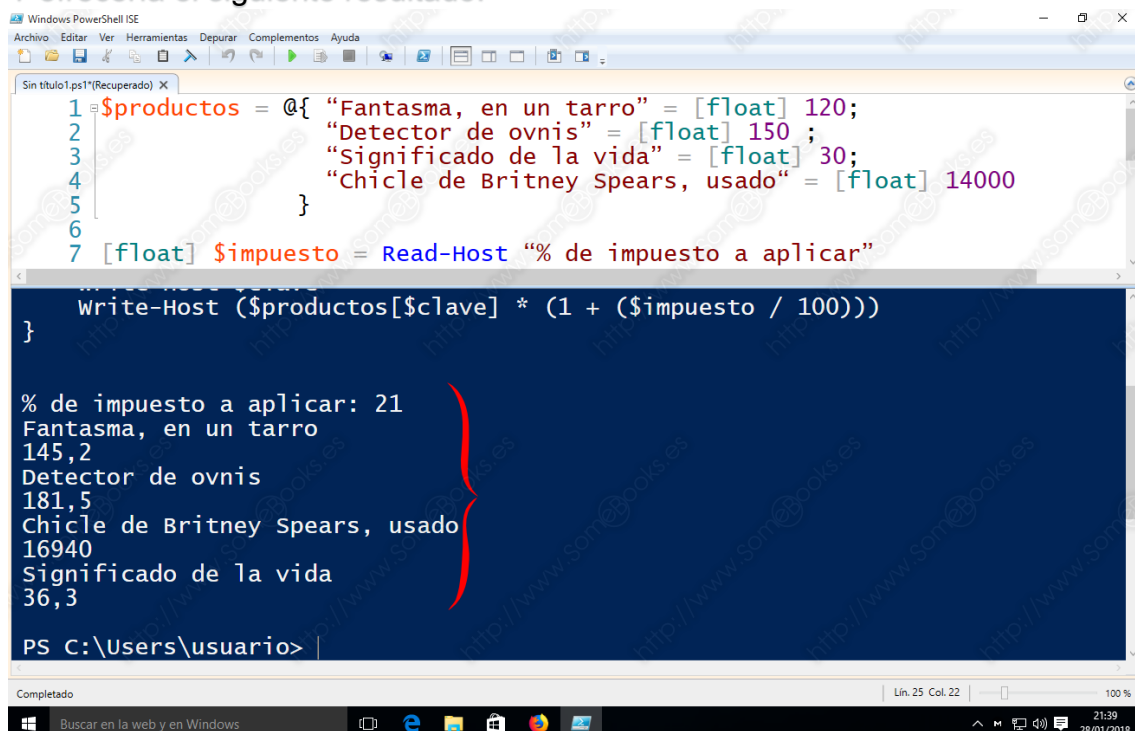
El *script* podría ser parecido a este:

```
$productos = @{"Fantasma, en un tarro" = [float] 120;
               "Detector de ovnis" = [float] 150 ;
               "Significado de la vida" = [float] 30;
               "Chicle de Britney Spears, usado" = [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)))
}
```

Y ofrecería el siguiente resultado:



```
Sin título1.ps1(Recuperado) X
1 $productos = @{"Fantasma, en un tarro" = [float] 120;
2               "Detector de ovnis" = [float] 150 ;
3               "Significado de la vida" = [float] 30;
4               "Chicle de Britney Spears, usado" = [float] 14000
5               }
6
7 [float] $impuesto = Read-Host "% de impuesto a aplicar"

Write-Host ($productos[$clave] * (1 + ($impuesto / 100)))
}

% de impuesto a aplicar: 21
Fantasma, en un tarro
145,2
Detector de ovnis
181,5
Chicle de Britney Spears, usado
16940
Significado de la vida
36,3

PS C:\Users\usuario>
```


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["Espíritu emprendedor"] = [float] 120
```

También puedes utilizar sintaxis orientado a objetos:

```
$productos.Add("Espíritu emprendedor", [float] 120)
```

O, incluso...

```
$productos."Espíritu emprendedor" = [float] 120
```

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

```
$productos.Remove("Espíritu emprendedor")
```

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

```
$productos.clear()
```