

# Introducción a Windows Power Shell – Parte II

<b>1. Estructuras condicionales</b>	<b>2</b>
Uso de la orden if	2
Uso de la orden if con else	3
Uso de la orden if con elseif	3
Uso de la orden switch	4
<b>2. Estructuras repetitivas</b>	<b>8</b>
La estructura do-while	9
La estructura while	10
La estructura do-until	10
La estructura for	11
La estructura foreach	13
Alterar el proceso de las estructuras repetitivas	14
La orden break	14
La orden continue	15

## 1. Estructuras condicionales

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**.

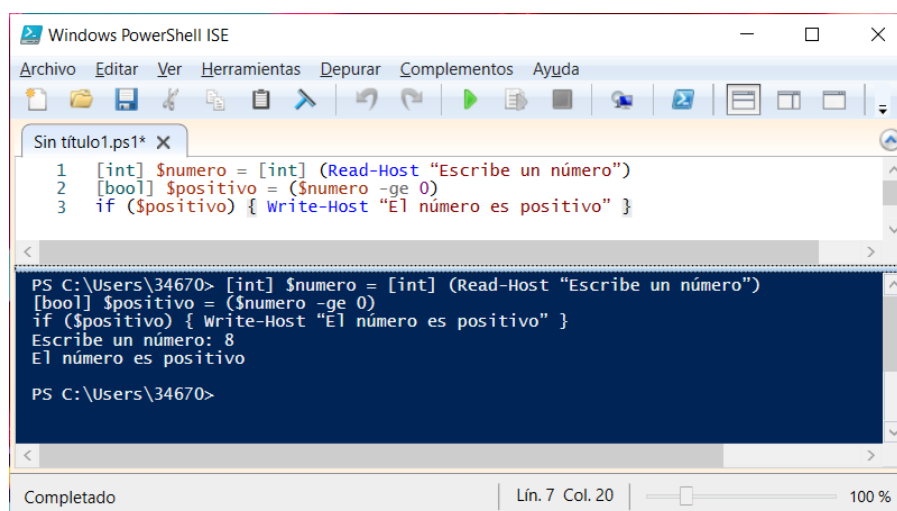
### Uso de la orden if

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 }
```

- 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.

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



```
Windows PowerShell ISE  
Archivo Editar Ver Herramientas Depurar Complementos Ayuda  
Sin título1.ps1* X  
1 [int] $numero = [int] (Read-Host "Escribe un número")  
2 [bool] $positivo = ($numero -ge 0)  
3 if ($positivo) { Write-Host "El número es positivo" }  
  
PS C:\Users\34670> [int] $numero = [int] (Read-Host "Escribe un número")  
[bool] $positivo = ($numero -ge 0)  
if ($positivo) { Write-Host "El número es positivo" }  
Escribe un número: 8  
El número es positivo  
  
PS C:\Users\34670>  
Completado | Lín. 7 Col. 20 | 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**. De hecho, **dentro del paréntesis podemos utilizar cualquier expresión válida que ofrezca un valor lógico**.

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

Otra opción que nos **ahorraría también la variable \$numero** sería la siguiente.

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

Todos los ejemplos anteriores son equivalentes. Cuando los ejecutemos, si escribimos un número positivo, el *script* te contestará con el texto *El número es positivo*. Y si escribimos 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 }

[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" }
```

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, podemos **utilizar tantos elseif como requiera el problema** que estés resolviendo. Por ejemplo, **necesitamos 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" }
```

Podríamos **simplificar las condiciones** si no diferenciamos entre minúsculas y mayúsculas. Otro de los inconvenientes es que pueden darse respuestas que no habíamos tenido en cuenta al principio. Por ejemplo, SI ó si, por lo que otra opción muy interesante sería **utilizar el método IndexOf**. 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.

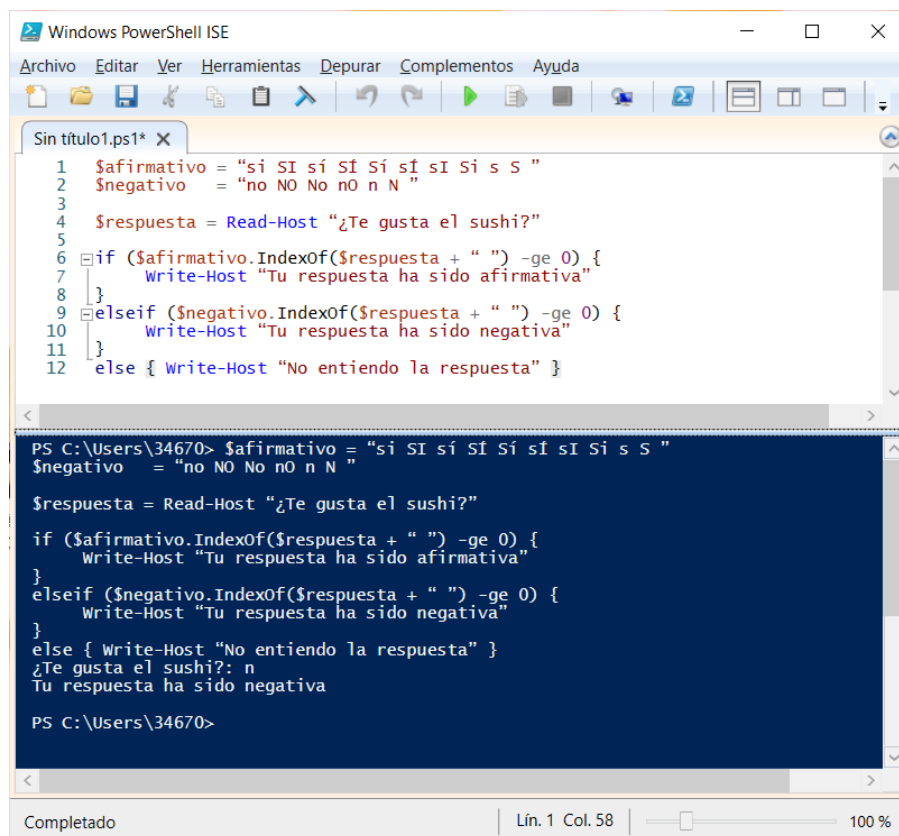
```

$afirmativo = "si SI sí SÍ Sí sí sI Si s S "
$negativo   = "no NO No nO n 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" }

```



```

Windows PowerShell ISE
Archivo Editar Ver Herramientas Depurar Complementos Ayuda

Sin título1.ps1* X
1 $afirmativo = "si SI sí SÍ Sí sí sI Si s S "
2 $negativo   = "no NO No nO n N "
3
4 $respuesta = Read-Host "¿Te gusta el sushi?"
5
6 if ($afirmativo.IndexOf($respuesta + " ") -ge 0) {
7     Write-Host "Tu respuesta ha sido afirmativa"
8 }
9 elseif ($negativo.IndexOf($respuesta + " ") -ge 0) {
10    Write-Host "Tu respuesta ha sido negativa"
11 }
12 else { Write-Host "No entiendo la respuesta" }

PS C:\Users\34670> $afirmativo = "si SI sí SÍ Sí sí sI Si s S "
$negativo   = "no NO No nO n 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" }
¿Te gusta el sushi?: n
Tu respuesta ha sido negativa

PS C:\Users\34670>

```

Completado | Lín. 1 Col. 58 | 100 %

## 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**. 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**.

```

switch (valor de prueba) {
    patrón 1 { bloque de código 1 }
    patrón 2 { bloque de código 2 }
    patrón n { bloque de código n }
    default { bloque de código por defecto }
}

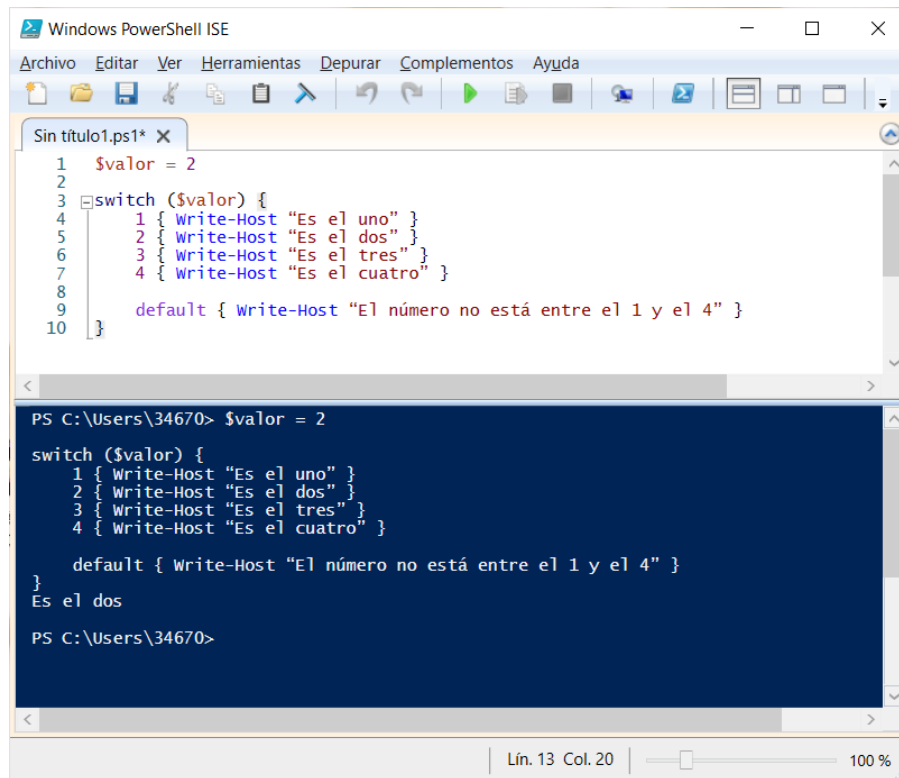
```

Además de la **pareja de llaves que engloba cada bloque de código**, existe una **pareja de llaves que engloba la estructura completa**.

```
$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 el 1 y el 4" }
}
```



Una de las características que **diferencian a switch de if** es que, en este caso, **siempre se evalúan todos los patrones**. Comprobamos que las condiciones no dejan de evaluarse después de la primera coincidencia:

```
$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" }

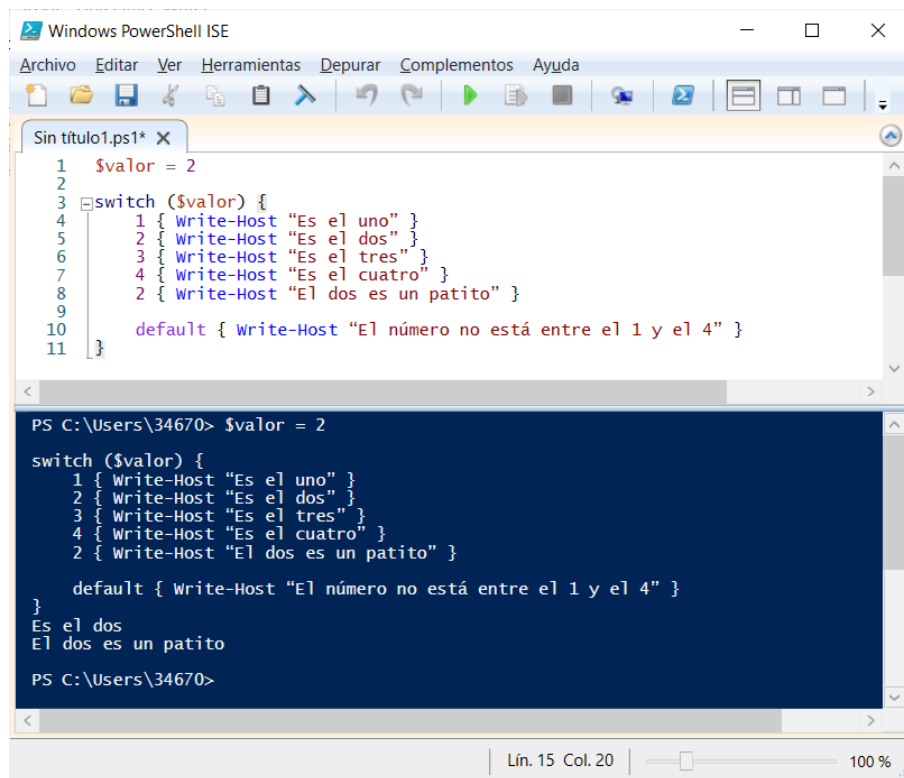
    default { Write-Host "El número no está entre el 1 y el 4" }
}
```

Incluso podemos evaluar diferentes variables o expresiones al mismo tiempo 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 el 1 y el 4" }
}
```



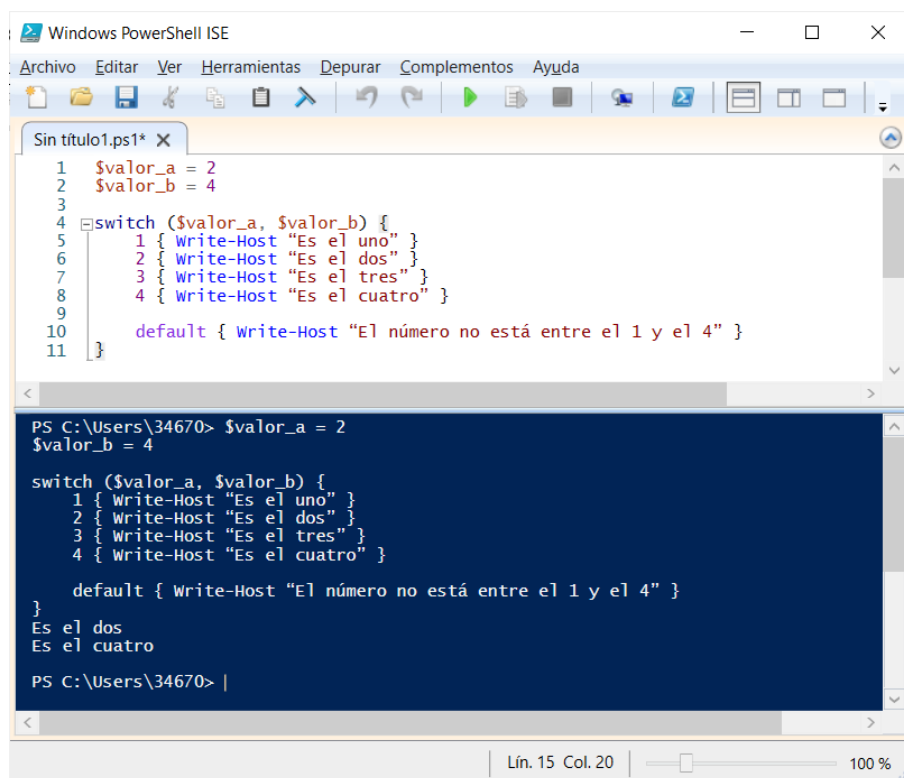
```
Sin título1.ps1* X
1 $valor = 2
2
3 switch ($valor) {
4     1 { Write-Host "Es el uno" }
5     2 { Write-Host "Es el dos" }
6     3 { Write-Host "Es el tres" }
7     4 { Write-Host "Es el cuatro" }
8     2 { Write-Host "El dos es un patito" }
9
10    default { Write-Host "El número no está entre el 1 y el 4" }
11 }
```

```
PS C:\Users\34670> $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 el 1 y el 4" }
}
Es el dos
El dos es un patito

PS C:\Users\34670>
```



```
Sin título1.ps1* X
1 $valor_a = 2
2 $valor_b = 4
3
4 switch ($valor_a, $valor_b) {
5     1 { Write-Host "Es el uno" }
6     2 { Write-Host "Es el dos" }
7     3 { Write-Host "Es el tres" }
8     4 { Write-Host "Es el cuatro" }
9
10    default { Write-Host "El número no está entre el 1 y el 4" }
11 }
```

```
PS C:\Users\34670> $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 el 1 y el 4" }
}
Es el dos
Es el cuatro

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

Observa que **PowerShell ha encontrado dos coincidencias** (una por cada variable). Cuando necesitemos 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 donde al evaluar el valor 2, **ejecuta la orden break** y se interrumpe la ejecución de switch.

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"; break }
    2 { Write-Host "Es el dos"; break }
    3 { Write-Host "Es el tres"; break }
    4 { Write-Host "Es el cuatro"; break }

    default { Write-Host "El número no está entre el 1 y el 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**.

```
[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" }
    5 { Write-Host "Suficiente" }
    6 { Write-Host "Bien" }
    { $_ -in 7, 8 } { Write-Host "Notable" }
    { $_ -in 9, 10 } { Write-Host "Sobresaliente" }

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

Dentro de la expresión, el **símbolo \$\_** está representado el resultado de evaluar la variable contenida en **switch**.

The screenshot shows the Windows PowerShell ISE interface. The script editor displays a switch statement that evaluates a grade (\$nota) and outputs a corresponding grade label. The console window shows the execution of the script, where the user enters '7' and the output is 'Aprobado'.

```
Sin título1.ps1* 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     { $_ -in 1..4 } { Write-Host "Insuficiente" }
7     5 { Write-Host "Suficiente" }
8     6 { Write-Host "Bien" }
9     { $_ -in 7, 8 } { Write-Host "Notable" }
10    { $_ -in 9, 10 } { Write-Host "Sobresaliente" }
11
12    default { Write-Host "No conozco esa nota." }
13 }

PS C:\Users\34670> [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" }
    5 { Write-Host "Suficiente" }
    6 { Write-Host "Bien" }
    { $_ -in 7, 8 } { Write-Host "Notable" }
    { $_ -in 9, 10 } { Write-Host "Sobresaliente" }

    default { Write-Host "No conozco esa nota." }
}
Escribe una nota: : 7
Aprobado
Notable

PS C:\Users\34670>
```

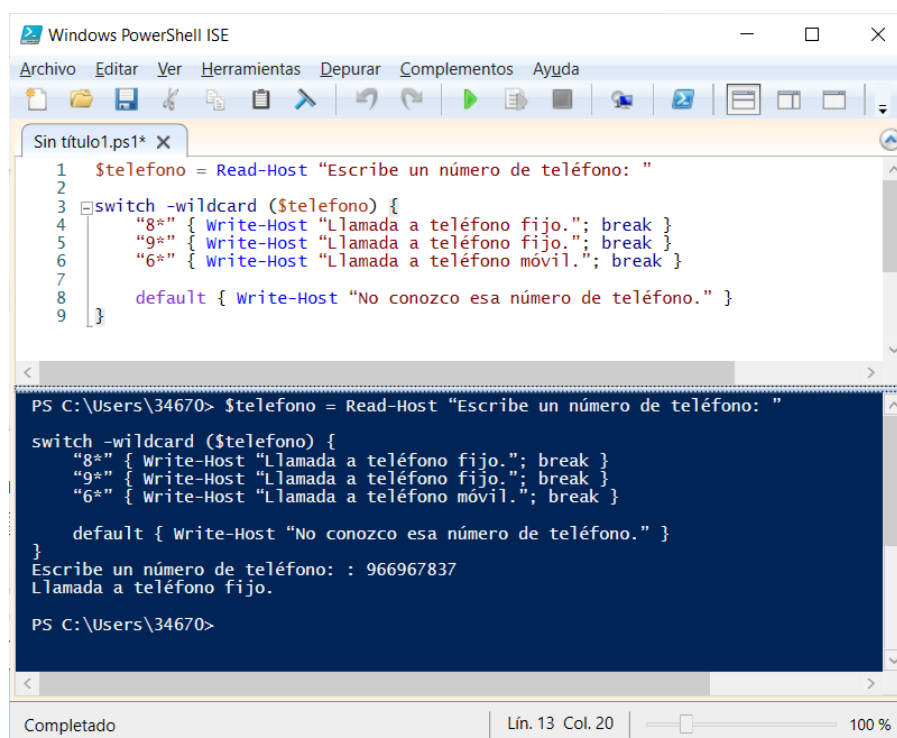
Por último, existen algunos **modificadores** que nos permiten actuar sobre la variable o expresión contenida en el switch:

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

```
$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 { Write-Host "No conozco esa número de teléfono." }
}
```



```
Windows PowerShell ISE
Archivo Editar Ver Herramientas Depurar Complementos Ayuda

Sin título1.ps1* 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 }
6     "6*" { Write-Host "Llamada a teléfono móvil."; break }
7
8     default { Write-Host "No conozco esa número de teléfono." }
9 }

PS C:\Users\34670> $telefono = Read-Host "Escribe un número de teléfono: "
switch -wildcard ($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 { Write-Host "No conozco esa número de teléfono." }
}
Escribe un número de teléfono: : 966967837
Llamada a teléfono fijo.

PS C:\Users\34670>
```

## 2. Estructuras repetitivas

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

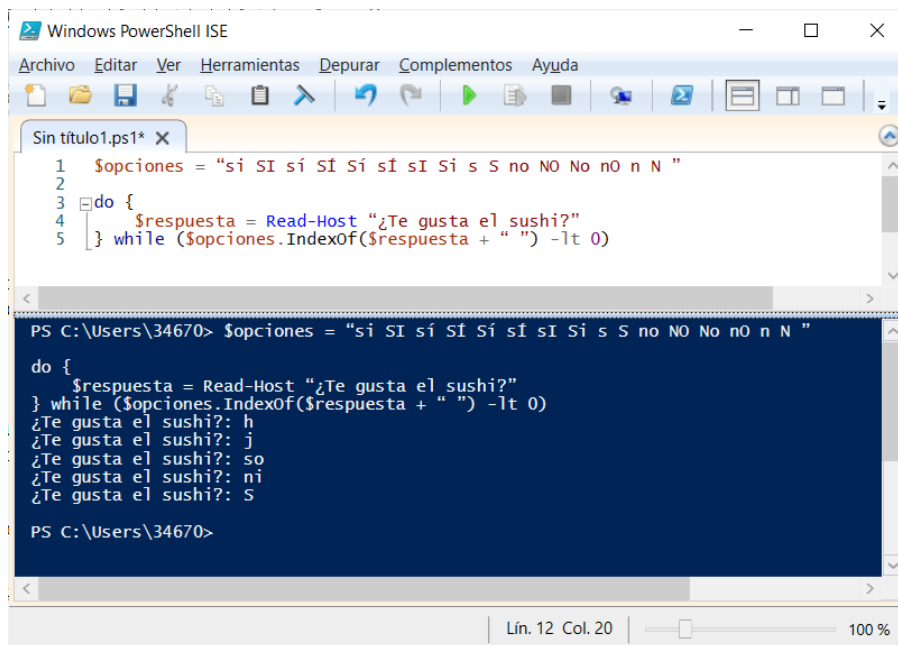
La estructura **do-while** nos permite repetir un bloque de código mientras una determinada condición siga ofreciendo el valor **\$true**. Su formato es como sigue:

```
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 Sí sí sI Si s S no NO No nO n N "
```

```
do {
    $respuesta = Read-Host "¿Te gusta el sushi?"
} while ($opciones.IndexOf($respuesta + " ") -lt 0)
```



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

```
$opciones = "sí SI sí SÍ Sí sí sI Si s S no NO No nO n 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)

```

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

```

1 $opciones = "si SI sí SÍ Sí sí sI Si s S no NO No nO n N "
2
3 do {
4     $respuesta = Read-Host "¿Te gusta el sushi?"
5
6     if ($opciones.IndexOf($respuesta + " ") -lt 0) {
7         Write-Host "Tu respuesta es errónea."
8     }
9 } while ($opciones.IndexOf($respuesta + " ") -lt 0)

```

The console window at the bottom shows the execution of the script:

```

PS C:\Users\34670> $opciones = "si SI sí SÍ Sí sí sI Si s S no NO No nO n 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)
¿Te gusta el sushi?: ho
Tu respuesta es errónea.
¿Te gusta el sushi?: is
Tu respuesta es errónea.
¿Te gusta el sushi?: S
PS C:\Users\34670>

```

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 el siguiente:

```
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 **while**:

```

$opciones = "si SI sí SÍ Sí sí sI Si s S no NO No nO n 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?"
}

```

## La estructura do-until

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 sintaxis es:

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

Su funcionamiento es idéntico a do-while, pero cambiando el sentido de la condición.

```
$opciones = "sí SI sí SÍ Sí sí sI Si s S no NO No nO n 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)
```

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 Sí sí sI Si s S no NO No nO n 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, vamos a ver qué significa cada uno 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.

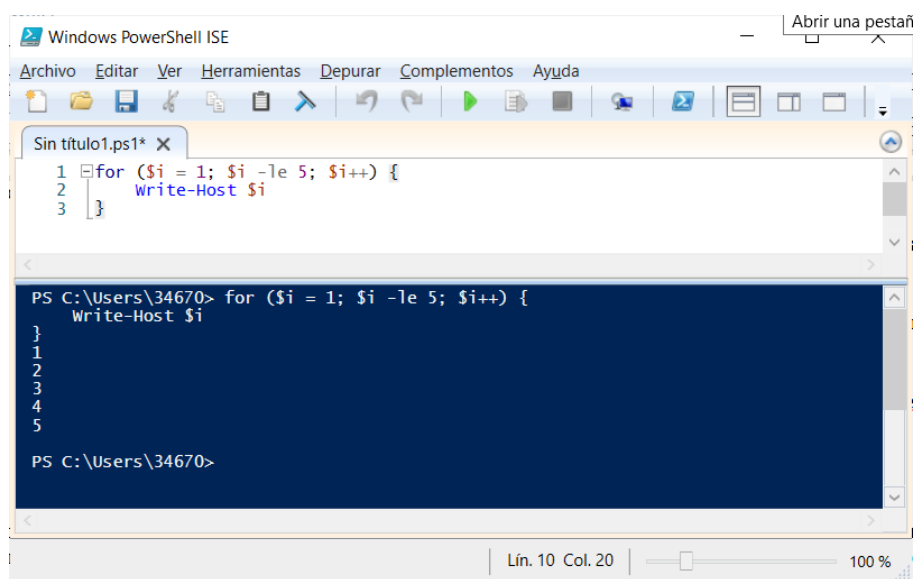
Veamos cómo funciona con un ejemplo, como mostrar en pantalla los números del 1 al 5:

```
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. Los argumentos de for van separados por caracteres de punto y coma (;).



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

```
1 for ($i = 1; $i -le 5; $i++) {  
2     Write-Host $i  
3 }
```

The console window at the bottom shows the execution of the script, displaying the numbers 1 through 5 on separate lines:

```
PS C:\Users\34670> for ($i = 1; $i -le 5; $i++) {  
    Write-Host $i  
}  
1  
2  
3  
4  
5  
PS C:\Users\34670>
```

Además, es muy sencillo **escribir un bucle while que se comporte como un bucle for**. El ejemplo anterior podríamos escribirlo de la siguiente forma y su funcionamiento será idéntico,

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

Basta con inicializar justo encima del bucle la variable y situar el incremento como última instrucción del bloque de código. Incluso, **podemos hacer que un bucle for se comporte como un bucle while**. Sería suficiente con **dejar vacíos la inicialización y el incremento**:

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

Vamos a ver un esquema que resume las principales características de las estructuras repetitivas.

	Mientras la condición sea <b>cierta</b>			Mientras la condición sea <b>falsa</b>
Formato	<b>for</b> (inicialización; condición; incremento) { bloque de código }	<b>while</b> (condición) { bloque de código }	<b>do</b> { bloque de código } <b>while</b> (condición)	<b>do</b> { bloque de código } <b>until</b> (condición)
Ejemplo	<b>for</b> (\$i = 1; \$i -le 10; \$i++) { Write-Host \$i }	\$i = 1  <b>while</b> (\$i -le 10) { Write-Host \$i \$i++ }	\$i = 1  <b>do</b> { Write-Host \$i \$i++ } <b>while</b> (\$i -le 10)	\$i = 1  <b>do</b> { Write-Host \$i \$i++ } <b>until</b> (\$i -gt 10)
	La condición se evalúa <b>al principio</b>			La condición se evalúa <b>al final</b>
	Si la condición <b>es falsa la primera vez</b> , el bloque de código no se ejecuta			El bloque de código se ejecuta siempre, <b>al menos la primera vez</b>

## La estructura foreach

Es una **estructura 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  
}
```

- **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**.

Como ejemplo vamos a ver cómo mostrar en pantalla los números del 1 al 5:

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

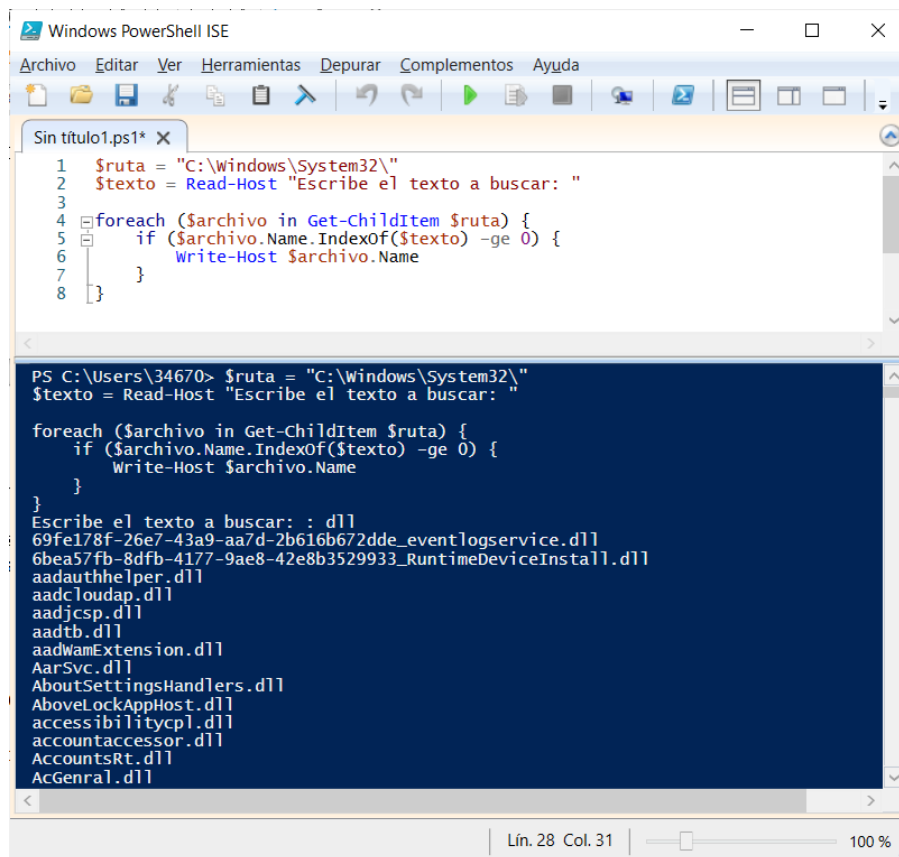
Cuando la lista es muy larga podemos utilizar también intervalos:

```
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., aunque en la imagen sólo veremos el principio de la lista, ya que es bastante larga.



```
Sin título1.ps1* X
1 $ruta = "C:\Windows\System32\"
2 $texto = Read-Host "Escribe el texto a buscar: "
3
4 foreach ($archivo in Get-ChildItem $ruta) {
5     if ($archivo.Name.IndexOf($texto) -ge 0) {
6         Write-Host $archivo.Name
7     }
8 }
```

```
PS C:\Users\34670> $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
    }
}
Escribe el texto a buscar: : dll
69fe178f-26e7-43a9-aa7d-2b616b672dde_eventlogservice.dll
6bea57fb-8dfb-4177-9ae8-42e8b3529933_RuntimeDeviceInstall.dll
aadauthhelper.dll
aadcloudap.dll
aadjcsp.dll
aadtb.dll
aadWamExtension.dll
AarSvc.dll
AboutSettingsHandlers.dll
AboveLockAppHost.dll
accessibilitycpl.dll
accountaccessor.dll
AccountsRt.dll
AcGenral.dll
```

## Alterar el proceso de las estructuras repetitivas

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

### La orden break

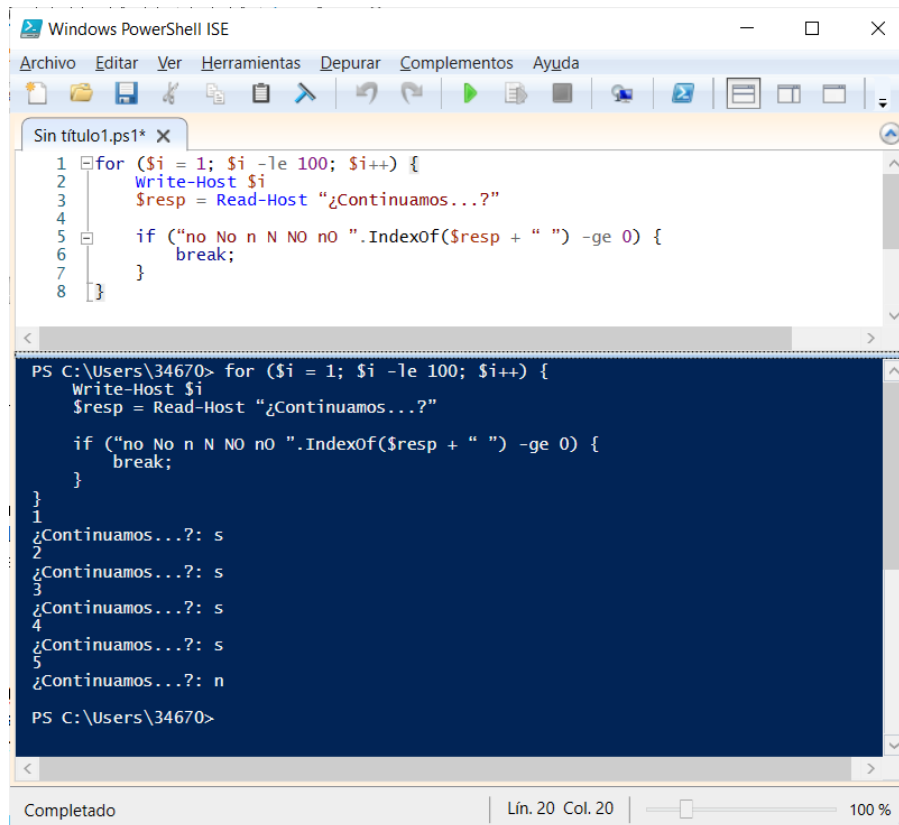
Al ejecutarla, PowerShell **finaliza la ejecución del resto de instrucciones del bloque de código y termina la repetición del bucle aunque no se haya cumplido la condición de salida**. La ejecución continuará por la siguiente instrucción después del bloque de código que se está repitiendo. En caso de no existir, finalizará 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 N NO nO ".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.



```

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

```

```

PS C:\Users\34670> for ($i = 1; $i -le 100; $i++) {
    Write-Host $i
    $resp = Read-Host "¿Continuamos...?"

    if ("no No n N NO nO ".IndexOf($resp + " ") -ge 0) {
        break;
    }
}
1
¿Continuamos...?: s
2
¿Continuamos...?: s
3
¿Continuamos...?: s
4
¿Continuamos...?: s
5
¿Continuamos...?: n
PS C:\Users\34670>

```

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."
}

```